UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Nikola Z. Vidič

PRIMENA MAŠINSKOG UČENJA U VERIFIKACIJI SOFTVERA

master rad

Beograd, 2019.

Mentor:

dr Milena Vujošević Janičić, docent
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Mladen Nikolić, docent
Univerzitet u Beogradu, Matematički fakultet

dr Filip Marić, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Naslov master rada: Primena mašinskog učenja u verifikaciji softvera

Rezime: Softver predstavlja sastavni deo svakodnevnog života i zahvaljujući napretku tehnologije pronalazi primenu u sve većem broju oblasti. Jedna od najvažnijih osobina softvera je njegova *ispravnost*. Ovde nije reč samo o jednostavnom pitanju da li softver radi ili ne. Prisustvo greške nekada može imati katastrofalne posledice. Na primer, greške u softveru koji se koristi u medicini mogu pacijente koštati života. Zato se posebna pažnja posvećuje *verifikaciji softvera* tj. ispitivanju njegove ispravnosti. Nažalost, verifikacija može biti vremenski i memorijski veoma skupa zbog čega je svako poboljšanje u efikasnosti jako važno. *Mašinsko učenje* se bavi proučavanjem procesa generalizacije - uopštavanja koje vodi ka generalnim zaključcima. Pomoću metoda mašinskog učenja moguće je konstruisati sisteme sposobne da poboljšaju svoje performanse na osnovu iskustva koje se ogleda u raspoloživim podacima. U ovom radu je pokušano da se uz pomoć mašinskog učenja dobije na efikasnosti u verifikaciji. U tu svrhu su pomoću metoda nadgledanog učenja konstruisani modeli čiji je zadatak da procene da li u nekom programu postoji greška i na taj način smanje potrebu za skupim verifikacijskim analizama.

Ključne reči: mašinsko učenje, verifikacija softvera, verifikacija, nadgledano učenje, klasifikacija

Sadržaj

Glava 1

Uvod

Živimo u dobu informacija, u svetu koji neprekidno i brzo tehnološki napreduje, u svetu neograničenih mogućnosti i prostora za napredak. Opravdanje za ovako smelu tvrdnju daje jedan mali, neobični uređaj bez koga prosečna osoba ne može da zamisli svoj dan - moderni mobilni telefon. Mogućnosti koje mobilni telefon pruža su ogromne. One prevazilaze jednostavno sredstvo komunikacije u vidu telefonskih razgovora i razmene poruka. Mobilni telefon takođe služi i kao budilnik, fotoaparat, video kamera, radio, muzički plejer, veb pregledač, uređaj za navigaciju. Spisak deluje impresivno, a impresivnije je da je pored svega nabrojanog pomoću mobilnog telefona moguće i transformisati uslikane fotografije i snimljene videe, gledati filmove i serije, kupiti namirnice, plaćati račune i mnogo toga još. Ali mobilni telefon je samo alat. U pozadini svih navedenih mogućnosti mobilnog telefona nalazi se jedna stvar, stvar koja je nestvarno učinila svakodnevnim - *softver*.

Softver čini da domen naučne fantastike od pre nekoliko vekova i decenija danas postaje realnost. Zahvaljujući softveru moguće je pre samog dolaska kući uključiti klima uređaj da ugreje ili rashladi kuću. Moguće je voziti se automobilom sposobnim da, bez ljudske pomoći, upravlja vozilom i parkira. Proboj na polju veštačke inteligencije omogućio je ličnog asistenta na uređaju koji stane u džep. Otkazivanje kamere prilikom fotografisanja mačke koja se nalazi u nezgodnom položaju ume da iznervira, ali ne može da se poredi sa otkazivanjem kočnica kada se ispred automobila nalaze pešaci. Zbog toga je jedan od najvažnijih aspekata softvera njegova ispravnost.

Informacije dolaze sa svih strana i čine veliki deo svakodnevnog života. Televi-

zija, novine, knjige, internet, obrazovne ustanove predstavljaju neke od uobičajenih izvora informacija. Softver je takođe izvor informacija, korišćenje softvera ostavlja trag u vidu informacija. Informacije dolaze u raznim oblicima i bilo da su to fotografije, zvučni ili video zapisi, bankovne transakcije, stavke računa iz prodavnice ili nešto drugo, informacija ima mnogo i njihov broj konstantno raste.

U ovom radu predstavljena je kombinacija dve važne oblasti, *verifikacije softvera* i *mašinskog učenja*. Za ispitivanje ispravnosti softvera koriste se metodi verifikacije softvera. Mašinsko učenje daje metode pogodne za sumiranje informacija i metode pogodne za obradu velike količine informacija.

Poglavlje 2 objašnjava šta je verifikacija softvera i razjašnjava njen značaj. Objašnjeni su pojmovi specifikacije (odeljak 2.1), verifikacije i validacije (odeljak 2.2) i detaljnije su objašnjene dve vrste verifikacije, dinamička (odeljak 2.3) i statička (odeljak 2.4).

Pregled mašinskog učenja dat je u poglavlju 3. Posebna pažnja posvećena je delu mašinskog učenja koji se zove nadgledano učenje (odeljak 3.1), kao i algoritmima nadgledanog učenja k najbližih suseda (odeljak 3.1.1), logistička regresija (odeljak 3.1.2) i slučajne šume (odeljak 3.1.3). U cilju razumevanja praktičnog dela rada opisane su korišćene tehnike pretprocesiranja podataka (odeljak 3.2). Objašnjeno šta je to matrica konfuzije (odeljak 3.3) i opisane su tehnike evaluacije klasifikacionih modela (odeljak 3.4), kao i izgled mašinskog učenja u programskom jeziku Python (odeljak 3.5).

Praktični deo rada opisan je u poglavlju 4. Detaljnije se diskutuje o podacima korišćenim za dobijanje klasifikacionog modela (odeljak 4.1), kao i samom procesu obučavanja klasifikacionog modela (odeljak 4.2). Pored toga u ovom poglavlju su opisani i program, koji na osnovu izvornih kodova programa napisanih u programskom jeziku C, računa atribute koji se koriste kao ulaz u prethodno obučeni model (odeljak 4.3) i primena modela na izračunate atribute (odeljak 4.4).

U poglavlju 5 sumirani su rezultati rada. Pored toga diskutovano je o pravcima daljeg razvoja.

Glava 2

Verifikacija softvera

Softver se nalazi svuda oko nas. U svakodnevnom životu, ljudi se sve više oslanjaju na sisteme zasnovane na informacionim i komunikacionim tehnologijama, pri čemu ti sistemi postaju sve složeniji i sve zastupljeniji. Ovi sistemi nalaze primenu u velikom broju oblasti: od berze, preko telefonskih i internet tehnologija pa sve do medicinskih sistema. Osim što se od njih očekuju dobre performanse u smislu vremena odziva i izračunavanja, jedan od ključnih aspekata kvaliteta sistema je odsustvo grešaka. Međutim, softver je pisan od strane ljudi pa stoga ne može biti savršen i podložan je greškama [18, 31].

Neke greške mogu biti pogubne po proizvođače. Greška pri deljenju brojeva u pokretnom zarezu na Intelovom Pentium II čipu (Intel Pentium Floating-Point Division Bug) ranih 1990-ih izazvala je gubitak od oko 475 miliona dolara da bi se čip zamenio i narušila je reputaciju Intela kao proizvođača pouzdanih čipova. Softverska greška u sistemu za upravljanje prtljagom prouzrokovala je odlaganje otvaranja aerodroma u Denveru za 9 meseci, što je dovelo do gubitka od 1,1 milion dolara po danu. Prestanak rada sistema za rezervaciju karata svake aviokompanije na svega 24 časa bi doveo kompaniju do bankrota zbog propuštenih porudžbina. NASA-ina kosmička letelica (NASA Mars Polar Lander) je usled nedovoljno testiranja 1999-te prilikom pokušaja sletanja prevelikom brzinom udarila u površinu Marsa i raspala se. Nekoliko stotina milijardi dolara potrošeno je za ispravljanje baga Y2K (The Y2K (Year 2000) Bug [16]) [18, 31].

Neke druge greške imaju daleko ozbiljnije posledice i njihova cena se ogleda u ljudskim životima. Softveri se koriste za upravljanje sistemima kao što su hemijske

i nuklearne elektrane, sistemima za upravljanje saobraćaja, sistemima za odbranu od olujnih talasa (eng. *storm surge barriers*), a greške ovih sistema mogu imati katastrofalne posledice. Greška u protivraketnom sistemu (Patriot Missile Defense System) 1991-ve dovela je do smrti 28 vojnika. Greška u mašini za terapiju radijacijom Therac-25 prouzrokovala je smrt šest pacijenata u periodu između 1985-te i 1987-me zbog prevelike izloženosti radijaciji [18, 31].

2.1 Specifikacija

Provera ispravnosti nekog softvera, odnosno njegovog ponašanja, nije moguća bez poznavanja željenog ponašanja datog softvera. Sa ciljem definisanja željenog ponašanja uvodi se pojam specifikacije. *Specifikacija* predstavlja dogovor razvojnog tima oko toga šta softver treba da radi.

Rezultat ispitivanja zahteva klijenta su podaci koji opisuju šta klijent očekuje od programa. Ovi podaci ne opisuju proizvod koji treba napraviti. Opis proizvoda se zadaje specifikacijom. Za formulisanje specifikacije koriste se informacije dobijene od klijenta ali specifikacija obuhvata i zahteve koje program mora da ispuni a koje klijent nije naveo. Preciznost specifikacije može da varira. Kompanije koje razvijaju proizvode u oblastima medicine, vazduhoplovstva, kao i za vladine organizacije koriste striktniji oblik specifikacije koji podrazumeva veliki broj provera. Rezultat je iscrpna i temeljna specifikacija koja se ne može menjati, osim pod ekstremnim okolnostima i svaki član razvojnog tima tačno zna šta treba da napravi. Sa druge strane, kompanije koje razvijaju programe čija je primena u manje rizičnim oblastima primenjuju neformalniji vid specifikacije ili je uopšte ne primenjuju. Prednost ovakvog pristupa pisanju specifikacije ogleda se u fleksibilnosti, ali je veliki nedostatak potencijalna neusklađenost oko ponašanja programa kao i to što se sve do završetka projekta ne zna kako će njegova finalna verzija izgledati [27, 31].

2.2 Verifikacija i validacija

Verifikacija i validacija su pojmovi koji se često međusobno prepliću a imaju različito značenje. *Verifikacija* (eng. *software verification*) je postupak potvrđivanja da softver zadovoljava specifikaciju. *Validacija* (eng. *software validation*) je postupak potvrđivanja da softver zadovoljava potrebe korisnika. Razlika između

ova dva slična pojma se može razjasniti na primeru problema sa Habl teleskopom (eng. *Hubble telescope*). Habl je reflektujući teleskop koji koristi veliko ogledalo da uveća objekte ka kojima je uperen. Kako je ogledalo pravljeno da se koristi u svemiru, jedini način da se testira je bio pažljivim merenjem svih atributa ogledala i njihovim poređenjem sa specifikacijom. Testovi su prošli uspešno i teleskop je lansiran u zemljinu orbitu aprila 1990-te. Ubrzo nakon što je teleskop počeo sa radom otkriveno je da su slike koje vraća mutne. Naknadnim analizama utvrđeno je da je postojala greška u specifikaciji. Gde je bio problem? Testiranje je potvrdilo da ogledalo zadovoljava specifikaciju tj. verifikacija je bila uspešna. Ono što testiranjem nije otkriveno je postojanje greške u samoj specifikaciji. Ogledalo nije zadovoljavalo zahteve korisnika zbog čega validacija, da je rađena, ne bi prošla. Cena ove greške je bila u novoj svemirskoj misiji 1993-će da bi se dodalo korektivno sočivo [31].

Uopšteno govoreći, program ne zadovoljava specifikaciju ako sadrži greške koje narušavaju funkcionalne i nefunkcionalne željene osobine programa. Funkcionalne osobine definišu koji su to željeni izlazi za zadate ulaze, a u nefunkcionalne osobine softvera spadaju vreme odziva, performanse i efikasnost. Bitna klasa grešaka koje mogu da naruše funkcionalne osobine softvera je klasa grešaka koje utiču na ispunjavanje bezbednosnih zahteva, npr. dovode do kraha programa ili utiču na performanse i efikasnost. Neki od primera su pokušaj deljenja nulom, pokušaj dereferenciranja NULL pokazivača, pokušaj čitanja sadržaja van granica rezervisane memorije i prisustvo kružnih blokada. Ove greške se mogu javiti nevezano od namene programa [27].

Osnovne vrste verifikacije softvera su dinamička i statička [27]. Ova dva pojma se slikovito mogu opisati na primeru kupovine polovnog automobila. Gledanje ispod haube, traženje tragova rđe, traženje ulubljenja, traženje ogrebotina na farbi, naprsnuća stakla, šutiranje guma su tehnike statičkog testiranja. Paljenje motora automobila i vožnja koja obuhvata dodavanje gasa i kočenje su tehnike dinamičkog testiranja [31].

2.3 Dinamička verifikacija softvera

Dinamička verifikacija softvera je provera njegove ispravnosti u toku izvršavanja. Provera ispravnosti se najčešće vrši testiranjem. Samim testiranjem se ne može dokazati ispravnost programa. Ono što testiranje može da uradi je da pokaže prisustvo

grešaka, ne i njihovo odsustvo. Zbog toga je svrha testiranja otkrivanje grešaka koje program sadrži [27].

Testiranje je proces u kojem se program pokreće sa reprezentativnim skupom ulaznih podataka, nakon čega se rezultati porede sa očekivanim. Strategija testiranja služi za određivanje reprezentativnih ulaza pri čemu reprezentativni skup treba da zadovoljava sledeće uslove: (1) ima visok potencijal otkrivanja greške, (2) relativno je male veličine, (3) vodi do visokog poverenja u softver nakog što softver uspešno prođe sve test primere. Dva osnovna izvora testova su programski kôd i specifikacija programa. U zavisnosti od izvora testova postoje tri strategije testiranja [27]:

Strategije testiranja crne kutije (eng. *black-box testing*) su strategije generisanja testova na osnovu specifikacije programa. Poznate su pod nazivima testiranje vođeno podacima ili testiranje vođeno ulazom i izlazom. Ideja je da se program posmatra kao crna kutija, sve se zasniva na zahtevima i specifikaciji programa, bez ikakvih briga o njegovoj strukturi i unutrašnjem ponašanju. Fokus strategije je na pronalaženju uslova koji dovode do ponašanja programa koje je suprotno ponašanju navedenom u specifikaciji. Test primeri se izvode samo na osnovu specifikacije softvera. U okviru ovih strategija izdvajaju se testiranje funkcionalnih i nefunkcionalnih osobina softvera, kao i regresiono testiranje [27, 29, 24].

Da bi se ovom strategijom pronašle sve greške koje postoje u programu potrebno je testirati svaki mogući ulaz u program.
Za testiranje programa koji rešava problem trouglova (za date celobrojne vrednosti stranica trougla program daje odgovor na pitanje da li je trougao jednakokraki, jednakostranični ili nejednakostranični [33]) potrebno je testirati sve validne i nevalidne trouglove. Skup celobrojnih vrednosti je sam po sebi ogroman, uz nevalidne ulaze broj ulaza koje treba proveriti se drastično povećava. Iscrpno testiranje velikih programa predstavlja jos veći problem. Da bi se testirala ispravnost kompajlera potrebno je kao ulaz koristiti ne samo svaki validni program (kojih ima beskonačno), već i svaki nevalidni program (kojih takođe ima beskonačno). Tek tada bi se sa sigurnošću moglo tvrditi da kompajler neće uspešno prevesti neispravan program. A ovo je samo jedna osobina kompajlera. Postoji veliki broj relevantnih osobina kompajlera koje bi takođe trebalo utvrditi na isti način (npr. da je zadržana semantika izvornog pro-

grama - ako u programu imamo $a + b$ da je to prevedeno u *add* a, b a ne u *sub* a, b)).

Stvari se dodatno komplikuju kod programa koji koriste neki oblik memorije poput aplikacija koje rade sa bazama podataka. U slučaju sistema za rezervaciju avionskih karata izvršavanje jedne transakcije (na primer, provera slobodnih mesta čitanjem iz baze, rezervacija karte upisom u bazu) zavisi od toga šta se desilo u prethodnim transakcijama. Stoga je potrebno proveriti ne samo sve moguće validne i nevalidne transakcije već i sve moguće sekvence transakcija [29].

Prethodni primeri pokazuju da je iscrpno testiranje nemoguće zbog beskonačnog skupa test primera. Pošto je prostor mogućih ulaza u program veliki, bira se njegov podskup. Veliki nedostatak ove strategije je i što osoba koja testira softver ne može da zna koliki je procenat softvera testiran. Takođe, postoji opasnost da neke putanje izvršavanja nikad ne podlegnu testiranju. Sa druge strane, ovakav pristup testiranja pomaže testeru da izabere konačne podskupove ulaza (test primera) na osnovu kojih će se pronaći što veći broj grešaka [29, 24].

Strategije testiranja bele kutije (eng. *white-box testing*) su strategije kod kojih se testiranje zasniva na poznavanju unutrašnjih putanja, strukture i implementacije koda. Informacije dobijene na osnovu gledanja koda programa koriste se da bi se donela odluka o tome šta testirati i kako pristupiti testiranju i sami testovi se generišu na osnovu programskog koda. Nazivaju se i strategijama strukturnog testiranja (eng. *structural testing*) zato što se testovi kreiraju i izvršavaju na osnovu posmatranja strukture koda. Najčešći primer ovakvog testiranja su jedinični testovi (testovi jedinica koda). Ideja ove strategije je da se napravi paralela sa strategijom crne kutije pri čemu se test primeri dobijaju iz strukture softvera. Osnovne strategije strukturnog testiranja zasnivaju se na kriterijumu pokrivenosti koda (eng. *code coverage*). Pokrivenost se može računati prema broju izvršenih putanja kroz program, broju instrukcija, broju grana nakon svake instrukcije grananja ili kombinacijom prethodnih kriterijuma. Smatra se da je pomenuta paralela sa strategijom crne kutije iscrpno testiranje putanja kroz program) [24, 31, 27, 29].

Slično strategijama crne kutije i ovde postoje nedostaci: (1) broj jedinstvenih logičkih putanja kroz program može biti ogroman i (2) čak iako se testiraju

sve putanje moguće je da program sadrži greške. Ovo se može desiti iz jednog od tri razloga: (1) iscrpno testiranje putanja ne daje garanciju da program zadovoljava specifikaciju (pri pisanju programa koji uređuje neke vrednosti u rastućem poretku moguće je napraviti grešku i uređenje vršiti u opadajućem poretku), (2) moguće je da program ne radi kako treba zato što mu nedostaje neka putanja (neke nedostajuće putanje testiranje može da detektuje, a neke ne može) i (3) greške osetljive na podatke (eng. *data sensitive errors*) mogu ostati neotkrivene (na primer, kod poređenja da li je razlika dva broja manja od neke vrednosti, računanje $a - b < c$ će sadržati greške zato što zapravo treba posmatrati apsolutnu vrednost razlike). Još jedan nedostatak je što osoba koja testira program, da bi isti razumela, mora da poseduje znanje iz oblasti programiranja. Prednost ove strategije je što se identifikuju i testiraju sve putanje kroz softver [27, 29, 24].

Strategije testiranja sive kutije (eng. *gray-box testing*) predstavljaju mešavinu prethodnih strategija. U ovim strategijama se na softver i dalje gleda kao na crnu kutiju s tim što se pored toga „zaviri" u kôd programa tj. koriste se i specifikacija i programski kôd [27, 31].

2.4 Statička verifikacija softvera

Statička verifikacija softvera je provera njegove ispravnosti na osnovu koda, bez njegovog izvršavanja. Analiza koda može biti od strane ljudi ili automatizovana.

Neodlučivost *halting problema* (problema ispitivanja ispravnosti programa) predstavlja ograničenje za automatizaciju provere ispravnosti programa. Posledica neodlučivosti je da se osobine polaznog sistema mogu verno opisati neodlučivim teorijama, ali proces dokazivanja nije moguće automatizovati baš zbog neodlučivosti teorija. Delimično rešenje ovog problema je u korišćenju interaktivnih dokazivača teorema (eng. *interactive theorem provers*) kao što su Isabelle i Coq. Korišćenjem ovih sistema moguće je pouzdano interaktivno dokazati razne osobine ispravnosti programa. Proverom samog dokaza dodatno se obezbeđuje najviši stepen pouzdanosti i sigurnosti u dokaz ispravnosti. Problem sa ovim sistemima je što zahtevaju ljudski rad zbog čega se ne koriste za čitave programe, već za njihove delove ili kompleksne osobine [27]. Potpuna automatizacija procesa verifikacije moguća je ako se polazni sistem opiše nekom odlučivom teorijom, ali često pod pretpostavkom neo-

graničenih resursa. Neograničeno vreme, kao i bilo koji drugi neograničeni resurs u praksi nisu mogući. Svaki konačan opis u okviru odlučive teorije je, u opštem slučaju, samo aproksimacija ponašanja polaznog sistema. Posledice aproksimacija su neprecizna analiza jer opis polaznog sistema nije veran, kao i analiza koju za neke programe nije moguće završiti zbog nedostatka resursa ili u konačnom vremenu. Kvalitet analize tj. dobijanje preciznijih rezultata u konačnom vremenu i sa konačnim resursima, može se poboljšati dodavanjem obeležja (eng. *annotations*). Obeležjima se dodaju informacije koje je teško ili nemoguće automatski izvesti. Takođe se dodaju i informacije koje je moguće automatski izvesti ali se dodavanjem tih informacija štedi na vremenu potrebnom za analizu. Obeležjima se najčešće zadaju invarijante petlji, kao i preduslovi i postuslovi funkcija [27]. Potpuno automatizovani pristupi verifikaciji programa su simboličko izvršavanje, apstraktna interpretacija i proveravanje modela.

2.4.1 Simboličko izvršavanje

Simboličko izvršavanje (eng. *symbolic execution*) predstavlja analizu programa u kojoj se umesto konkretnih prate simboličke vrednosti promenljivih. Ovakvom analizom se kreiraju simbolički izrazi koji opisuju određene putanje kroz program što omogućava istovremenu proveru svih ulaza koji prate istu putanju. Simboličko izvršavanje istovremeno ispituje veliki broj putanja izvršavanja bez potrebe za konkretnim ulaznim parametrima. Umesto toga, vrši se apstrakcija ulaznih vrednosti simbolima nakon čega se koriste rešavači ograničenja (eng. *constraint solvers*) da bi se proizvele konkretne vrednosti ulaza koje dovode do grešaka. Simboličko izvršavanje je nastalo sredinom 1970-ih sa ciljem otkrivanja da li se u okviru softvera mogu narušiti određena svojstva, npr. deljenje nulom, pokušaj dereferenciranja NULL pokazivača... Kada se testiranjem uz korišćenje konkretnih vrednosti ulaza prođe kroz jednu putanju programa, ona je izvršena samo za te konkretne vrednosti. Nasuprot tome, kada se simboličkim izvršavanjem prođe kroz jednu putanju programa, ona je izvršena za sve moguće vrednosti koje prolaze kroz tu putanju. Zbog toga je ključna ideja omogućiti programu da koristi simboličke vrednosti [27, 19].

Ako simboličko izvršavanje pokaže da neka putanja zadovoljava uslove ispravnosti programa tada svi ulazi koji prolaze putanju zadovoljavaju uslove programa čime se znatno dobija na efikasnosti provere u odnosu na tehnike dinamičke analize. Problem sa ovim pristupom je broj mogućih putanja koji je često prevelik da bi

se u potpunosti pretražio. Zbog toga simboličko izvršavanje svoju primenu češće nalazi u pronalaženju grešaka nego u verifikaciji programa. Veoma važan faktor koji utiče na brzinu otkrivanja greške u programu je redosled ispitivanja putanja. Ako se greška javlja u putanji koja se ispituje prva vreme otkrivanja greške je minimalno. U slučaju da se greška javlja u putanji koja se poslednja analizira vreme otkrivanja greške će biti veliko, a moguće je i da se greška uopšte ne pronađe. Ako postoji veliki broj putanja kroz program može doći do isteka vremena predviđenog za analizu pre nego što se greška otkrije [27].

2.4.2 Apstraktna interpretacija

„*Apstraktna interpretacija* (eng. *abstract interpretation*) tehnika je aproksimacije formalne semantike programa, tj. matematičkog modela ponašanja programa [27]". „Apstraktna interpretacija obuhvata procenu ponašanja programa na nekom apstraktnom domenu, sa ciljem dobijanja približnog rešenja [25]". Apstraktni domen je aproksimacija reprezentacije skupova konkretnih vrednosti, a za slikanje konkretnih vrednosti u vrednosti apstraktnog domena koristi se funkcija apstrakcije [25]. Semantika programa je opisana domenom D_c i relacijama nad domenom pri čemu se relacije mogu menjati tokom izvršavanja naredbi programa. Provera da li neko svojstvo važi nad domenom D_c može biti problem u slučaju velikih programa jer veliki programi znače velike domene i veliki broj mogućih putanja kroz program. Dodatan problem predstavlja neodlučivost koja se može javiti u raznim kontekstima. Kao potencijalno rešenje nameće se aproksimacija domena D_c apstraktnim domenom D_a. Apstraktni domen ne može biti precizan kao konkretni domen, ali može dati informacije o nekim svojstvima domena. Sa druge strane, apstrakcijom domena se mogu izgubiti važne informacije. Primer jedne apstrakcije bila bi apstrakcija beskonačnog domena celih brojeva koji se menja skupom znakova brojeva +, -, 0 . Ovakva apstrakcija bi verno prikazala znak rezultata množenja dva cela broja ako su poznati samo znaci operanada. Ako je informacija od značaja znak rezultata sabiranja dva cela broja, pri čemu su poznati samo znaci operanada, ovakva apstrakcija se u nekim slučajevima pokazuje kao nedovoljna (znak rezultata sabiranja brojeva -3 i 5 je (+), a znak rezultata sabiranja brojeva -7 i 5 (-)) [27].

2.4.3 Proveravanje modela

Proveravanje modela (eng. *model checking*) je metod verifikacije u kojem se sistem, hardverski ili softverski, koji je potrebno verifikovati opisuje konačnim aparatom (stanjima i tranzicijama između stanja), a specifikacija se zadaje u terminima temporalne logike (specifikacija je opisana kao logička formula). U temporalnoj logici je moguće razmatrati uređenje događaja u toku vremena. Na primer, moguće je da trenutno netačno svojstvo postane tačno u nekom trenutku u budućnosti. Stanje modela čine vrednosti promenljivih i stanja steka i hipa, a tranzicije opisuju kako program prelazi iz jednog stanja u drugo stanje. Metodi zasnovani na proveravanju modela ispituju stanja programa do kojih se može doći. Nakon što je sistem opisan, stanja automata se obilaze sa ciljem dokazivanja uslova specifikacije. Proveravanje modela metodom grube sile istražuje sva moguća stanja sistema na sistematičan način, slično programu za igranje šaha. Na ovaj način je moguće dokazati da sistem zadovoljava neko svojstvo. Međutim, sa trenutno raspoloživim procesorima i memorijama ispitivanje najvećih mogućih prostora stanja predstavlja izazov. Najbolji programi za proveravanje modela mogu da rade sa prostorima stanja reda 10^8 do 10^9, a korišćenjem pametnih algoritama i specijalnih struktura za konkretne probleme prostor stanja može drastično da se uveća (10^{20} do 10^{476}!). Metod proverava da li sva relevantna stanja sistema zadovoljavaju željena svojstva. Ako je prostor stanja konačan metod će se izvršiti u konačnom vremenu. U slučaju neuspeha tj. ako se pronađe stanje koje narušava svojstvo koje se razmatra, generiše se kontraprimer koji definiše putanju izvršavanja koja vodi od inicijalnog stanja sistema do stanja kojim je svojstvo koje se proverava narušeno. Mana proveravanja modela je eksplozivan porast stanja tj. broj mogućih stanja eksponencijalno raste sa porastom broja promenljivih (komponenti sistema). Zbog toga se koristi metod proveravanja ograničenih modela [27, 25, 23, 18].

Proveravanje ograničenih modela (eng. *bounded model checking*) je formalna tehnika verifikacije koja se mahom koristi u industriji poluprovodnika. Proveravanje ograničenih modela vrši brzu pretragu prostora stanja i za određene probleme daje znatno poboljšanje performansi izračunavanja u odnosu na metod proveravanja modela. Uspeh tehnike zasnovan je na velikim mogućnostima iskaznih SAT rešavača (eng. *propositional SAT solvers*). Ovaj metod se zasniva na ograničavanju dužine putanje stanja koja se proverava. Ako se za zadatu vrednost dužine ne pronađe greška moguće je povećati vrednost tako da se provera nastavlja dok se ne pronađe

greška, model ne postane prevelik za analizu ili vrednost dužine ne dostigne gornju granicu dužine mogućih putanja čime je model verifikovan [27, 25, 23].

Glava 3

Mašinsko učenje

Sama ideja *mašinskog učenja* (eng. *machine learning*) javlja se još četrdesetih godina dvadesetog veka u radovima Alana Tjuringa (Alan Turing). Razvoj mašinskog učenja vođen je željom da se razume i oponaša ljudski potencijal za učenje. Pedesetih godina mašinsko učenje se razvija zajedno sa pojmom perceptrona, pretka neuronskih mreža. Razvoj mašinskog učenja u formi neuronskih mreža nastavlja se, uz uspone i padove interesovanja, sve do danas. Početkom dvehiljaditih dešava se proboj na polju razvoja *veštačke inteligencije* (eng. *artificial inteligence*) i mnogi problemi za koje se smatralo da će jos dugo ostati nerešeni bivaju rešeni, velikim delom zahvaljujući mašinskom učenju. Uopšteno, „mašinsko učenje predstavlja disciplinu koja se bavi konstrukcijom sistema koji se prilagođavaju i popravljaju svoje performanse sa povećanjem iskustva, oličenog u količini relevantnih podataka [26]". Mašinsko učenje proučava indukcijski način zaključivanja tj. generalizaciju (uopštavanje ka univerzalnim zaključcima). Primene mašinskog učenja su brojne: prepoznavanje različitih oblika na slikama (na primer, lica i tumora), autonomno upravljanje vozilima (na primer, automobilima ili letelicama), igranje igara na tabli kao što je šah, klasifikacija teksta, prepoznavanje cifara i mnoge druge [26].

Mašinsko učenje za rešavanje problema koristi različite metode. Ove metode se prema prirodi problema učenja svrstavaju u jednu od tri grupe: *nadgledano učenje* (eng. *supervised learning*), *nenadgledano učenje* (eng. *unsupervised learning*) ili *učenje potkrepljivanjem* (eng. *reinforcement learning*). U nastavku teksta detaljnije su opisane neke tehnike nadgledanog učenja. Detaljnije informacije o nenadgledanom učenju i o učenju potkrepljivanjem nisu potrebne za razumevanje teze i mogu se naći u literaturi [22, 35, 26].

3.1 Nadgledano učenje

Nadgledano učenje se odlikuje datim vrednostima ulaza, ali i datim vrednostima izlaza koji im odgovaraju. Na osnovu tih podataka potrebno je odrediti vezu koja postoji između ulaza i izlaza. Nadgledano učenje se može posmatrati kao vid učenja u kom se znanje stiče iskustvom. Iskustvo je sadržano u podacima za treniranje, a stečeno znanje se primenjuje na nove podatke. Naziv je posledica sličnosti postupka nadgledanog učenja i učenja u kome profesor zada učeniku zadatke i nakon što ih učenik reši, dâ učeniku odgovore radi poređenja rezultata. Algoritmi mašinskog učenja se opisuju kao učenje ciljne funkcije f koja najbolje opisuje vezu između ulaznih promenljivih x i izlazne promenljive y, odnosno $y = f(x)$. Naučena veza (ciljna funkcija f) se kasnije koristi za buduća predviđanja izlaza y na osnovu novih vrednosti ulaza x. Najčešće je ulaz predstavljen vektorom vrednosti promenljivih koje se nazivaju *atributima* (eng. *features*), a izlaz kao jedna promenljiva koja se zove *ciljna promenljiva* (eng. *target variable*). Kako se u današnje vreme raspolaže ogromnim količinama podataka merenim gigabajtima i terabajtima, neophodno je pronaći metode koje automatski pronalaze veze između promenljivih. Veze tj. izgrađene ciljne funkcije se nazivaju *modelima mašinskog učenja*. Postoji veliki broj modela i ne opisuju svi podjednako dobro veze među podacima. Od kvalitetnog modela se očekuje da vrši dobru generalizaciju tj. da prilikom predviđanja retko greši [26, 22, 34].

Osnovne vrste nadgledanog učenja su *klasifikacija* i *regresija*. „Klasifikacija je problem predviđanja kategoričke ciljne promenljive [26]". Vrednosti kategoričke promenljive pripadaju nekom konačnom skupu, pri čemu ne postoji uređenje među tim vrednostima. „Regresija je problem predviđanja neprekidne ciljne promenljive [26]". Neprekidne promenljive uzimaju vrednosti iz neograničenog skupa vrednosti.

Preprilagođavanje modela Model klasifikacije može imati dve vrste grešaka: greške u fazi treniranja i greške u generalizaciji. Greška u fazi treniranja modela predstavlja broj grešaka usled pogrešne klasifikacije trening instanci. Greška u generalizaciji predstavljaju očekivanu grešku koju model pravi prilikom određivanja klasa nepoznatih instanci. Odlike dobrog modela su prilagodljivost trening podacima i tačno predviđanje klasa novih instanci. To znači da dobri modeli imaju male vrednosti grešaka i u fazi treniranja i u fazi generalizacije. Model koji previše dobro odgovara trening podacima, odnosno

ima malu grešku u fazi treniranja, može imati veću grešku u generalizaciji u odnosu na model sa većom greškom u fazi treniranja. Takva situacija se naziva *preprilagođavanje modela* (eng. *model overfitting*). Preprilagođavanje može nastati: (1) zbog nedostatka reprezentativnih uzoraka - model će doći do pogrešnih zaključaka zbog nedovoljnog broja trening podataka. (2) kao posledica prisustva šuma (eng. *noise*) među trening podacima. Na primer, nova instanca će biti pogrešno klasifikovana zato što su vrednosti nekih njenih atributa identične vrednostima atributa pogrešno klasifikovane trening instance. (3) kod modela čiji algoritmi učenja primenjuju proceduru višestrukog poređenja (eng. *multiple comparison procedure*). Neka je T_0 inicijalno drvo odlučivanja a T_x drvo nakon što se u njega doda unutrašnji čvor za atribut x. Dodavanjem novog čvora ne dobija se obavezno bolje drvo tj. bolji model. Čvor se može dodati u stablo ako je dobit $\Delta(T_0, T_x)$ veća od nekog zadatog parametra α. Ovaj način dodavanja čvorova u stablo je zavisan od funkcije podele Δ i parametra α. Ako postoji samo jedan uslov testiranja atributa izbor dovoljno velike vrednosti parametra α eliminiše dodavanje lažno najboljih čvorova podele (eng. *spurious nodes*) u stablo. Lažno najbolji čvorovi podele zadovoljavaju uslov $\Delta(T_0, T_{x_{max}}) > \alpha$ ali se njihovim dodavanjem ne dobija bolji model. Realnost je da u praksi postoji više uslova testiranja atributa, a drvo odlučivanja treba da izabere najbolji atribut podele x_k iz skupa atributa $\{x_1, x_2, ..., x_k\}$ testiranjem uslova $\Delta(T_0, T_{x_{max}}) > \alpha$. Sa većim brojem atributa za razmatranje raste i verovatnoća da će se pronaći atribut koji zadovoljava uslov $\Delta(T_0, T_{x_{max}}) > \alpha$. U ovoj situaciji algoritam može dodavati u stablo lažno najbolje čvorove podele, što vodi preprilagođavanju modela. Zato treba modifikovati funkciju dobiti Δ ili parametar α tako da u obzir uzimaju broj atributa k [36].

U nastavku će detaljnije biti opisani modeli klasifikacije zasnovani na algoritmima k najbližih suseda, slučajnim šumama i logističkoj regresiji.

3.1.1 Algoritam k najbližih suseda

Klasifikatori zasnovani na algoritmu k najbližih suseda spadaju u klasifikatore zasnovane na instancama. Kod klasifikatora zasnovnih na instancama faza treniranja se odlaže sve do poslednjeg koraka procesa klasifikacije. Za takve klasifikatore

se kaže da „lenjo" uče (eng. *lazy learners*). Najjednostavniji opis logike učenja zasnovanog na instancama je da slične instance imaju slične oznake klasa („ako hoda kao patka, priča kao patka i izgleda kao patka, onda je najverovatnije patka [36]") [17].

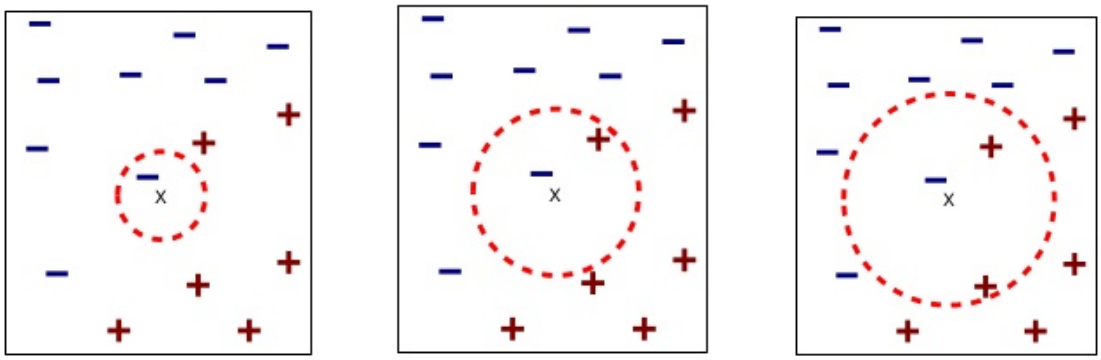
Reprezentacija modela koji koriste algoritam k najbližih suseda je čitav skup trening podataka. Pošto nema eksplicitnog modela, nema ni potrebe za učenjem istog. Trening podaci se skladište u memoriji pa je poželjno njihovo redovno održavanje (ažuriranje kada se dođe do novih podataka, brisanje podataka koji sadrže greške, brisanje odudarajućih podataka). Modeli svoja predviđanja vrše na osnovu trening podataka. Sva izračunavanja vrše se u fazi predviđanja. Ovi klasifikatori svoju predikciju zasnivaju na lokalnim podacima pa su stoga podložni greškama zbog postojanja šuma. Zavisni su od izbora mere bliskosti i adekvatnog preprocesiranja podataka. Ako se vrednosti atributa mere na različitim skalama, jedan atribut može više uticati na ishod klasifikacije a da to nije opravdano. Ovo se može rešiti svođenjem atributa na istu skalu standardizacijom. Pored svoje jednostavnosti, ovi modeli nalaze široku primenu [22, 26, 36].

3.1.1.1 Dodeljivanje klase objektu

Prilikom klasifikacije nepoznatog objekta prvo se među poznatim objektima pronađe njegovih k najbližih suseda na osnovu izabrane mere bliskosti (često je u upotrebi *Euklidsko rastojanje* ako su vrednosti atributa realni brojevi). Algoritam novom objektu pridružuje klasu koja se *najčešće javlja među njegovim susedima* (eng. *majority voting*). U slučaju nerešenog ishoda, klasa nepoznatog objekta se dobija slučajnim izborom iz skupa najzastupljenijih klasa. Da bi se izbegli nerešeni ishodi, česta je praksa da se za k uzima neparan broj [36, 26, 22].

Na slici 3.1 prikazana su susedstva instance koja se nalazi u sredini kruga kada algoritam k najbližih suseda uzima vrednosti 1, 2 i 3 za k. Instanci će biti dodeljena klasa na osnovu klasa njenih najbližih suseda. Na slici 3.1a klasa najbližeg suseda je (-) pa će i klasa instance biti (-). U slučaju 3.1c dva suseda su klase (+), a jedan klase (-) pa će klasa instance biti (+). Slika 3.1b ilustruje situaciju podjednake zastupljenosti klasa (jedan (+) i jedan (-)). U ovakvim situacijama, instanci se dodeljuje klasa slučajnim izborom [36].

Za rezultat klasifikacije jako je bitan izbor parametra k. U slučaju male vrednosti



(a) k=1 (b) k=2 (c) k=3

Slika 3.1: Susedstva i dodeljivanje klasa novoj instanci za k=1, 2 i 3

parametra k može doći do preprilagođavanja. Do grešaka dolazi jer je mali broj suseda uključen u razmatranje. Često se može javiti greška zbog prisustva šuma. Sa druge strane, velika vrednost parametra k vodi ka *potprilagođavanju* (eng. *model underfitting*) - situaciji kada model loše klasifikuje i trening i test podatke zato što nije uspeo da nauči pravu strukturu podataka sa kojima radi. U tom slučaju do greške može doći jer se razmatraju klase onih objekata koji nisu u neposrednom susedstvu. Izbor vrednosti parametra k je, u praksi, vođen nekom heuristikom. Čest način izbora parametra je poređenjem tačnosti modela konstruisanih za različite vrednosti parametra k [26, 36, 17].

3.1.2 Logistička regresija

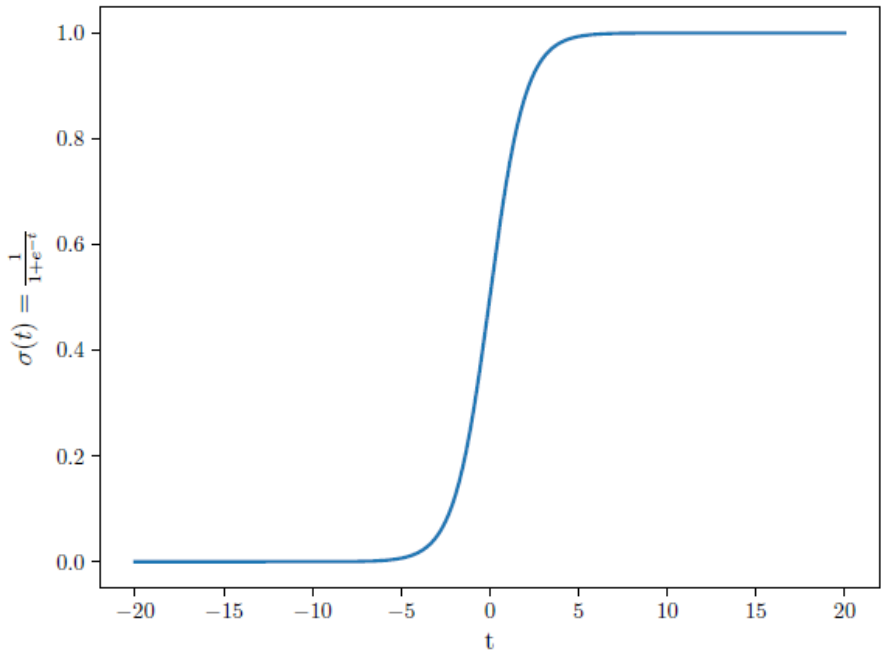
Logistička regresija predstavlja probabilistički model klasifikacije. Koristi se i za probleme binarne i za probleme višeklasne klasifikacije. Kod probabilističkih modela je potrebno definisati raspodelu verovatnoće koju kasnije treba oceniti na osnovu podataka. Da bi ocena raspodele bila računski izvodljiva potrebno je uvesti pretpostavke o: (1) raspodeli podataka i (2) međusobnoj zavisnosti promenljivih. Pogrešne pretpostavke o podacima, njihovoj raspodeli i zavisnosti, vode ka lošijim rezultatima predviđanja. U daljem tekstu detaljnije je opisana binarna logistička regresija [26].

Logistička regresija pretpostavlja međusobnu nezavisnost vrednosti ciljne promenljive y za date vrednosti atributa x i *Bernulijevu raspodelu* ciljne promenljive y za date vrednosti atributa x. To znači da postoji parametar μ iz intervala [0,1]

takav da je $p(y{=}1|x) = \mu$, a $p(y{=}0|x) = 1\text{-}\mu$. Ovako definisan model logističke regresije nije kompletan. Potrebno je definisati zavisnost parametra μ od vrednosti atributa x. Da bi se parametar μ uklopio u definiciju verovatnoće, njegove vrednosti moraju biti u intervalu [0,1]. Poželjno bi bilo koristiti model linearne regresije zbog njegove jednostavnosti i lake interpretabilnosti. Međutim, na prvi pogled korišćenje linearnog modela nije moguće jer vrednosti linearnog modela pripadaju intervalu $[-\infty, \infty]$. Korišćenje modela linearne regresije je moguće ako se njegove vrednosti transformišu nekom nenegativnom, monotonom, neprekidnom i diferencijabilnom funkcijom u interval [0,1]. U tu svrhu koristi se *sigmoidna funkcija* σ (moguća je upotreba i nekih drugih funkcija):

$$\sigma(t) = \frac{1}{1 + e^{(-t)}}$$

Sigmoidna (logistička) funkcija, čiji je grafik prikazan na slici 3.2, uzima proizvoljan realan broj i dodeljuje mu vrednost iz intervala (0,1).



Slika 3.2: Sigmoidna funkcija

Transformacijom vrednosti linearnog modela, definisana je i zavisnost parametra Bernulijeve raspodele μ od vrednosti atributa x. Slično modelu linearne regresije, forma modela logističke regresije je jednačina:

$$p_w(y = 1|x) = \sigma(w \cdot x)$$

Izlaz modela je linearna kombinacija njegovog ulaza i koeficijenata. Koeficijenti se dobijaju iz podataka prilikom samog treniranja modela i oni predstavljaju reprezentaciju modela u memoriji [26, 22].

Treniranje modela linearne regresije odgovara oceni njegovih parametara i zasniva se na **principu maksimalne verodostojnosti**. Kao što smo videli, da bi se precizirao statistički model potrebno je uključiti neke parametre. Da bi dobijeni rezultat imao smisla, izbor vrednosti parametara je jako bitan. Jedan od principa izbora ovih vrednosti, princip maksimalne verodostojnosti, je prihvatanje onih vrednosti parametara za koje su posmatrani podaci visoko verovatni [26].

3.1.3 Slučajne šume

Ansambl metodi (eng. *ensemble methods*) su tehnike koje za cilj imaju poboljšanje tačnosti klasifikacije koje se postiže kombinovanjem predviđanja većeg broja klasifikatora. Ovi metodi konstruišu veći broj *baznih klasifikatora* (eng. *base classifiers*) i kombinovanjem njihovih rezultata vrše svoje predviđanje. Rezultat predviđanja je srednja vrednost u slučaju regresije ili najzastupljenija vrednost u slučaju klasifikacije. Rezultati dobijeni primenom ansambl klasifikatora mogu biti bolji od rezultata dobijenih korišćenjem samo jednog klasifikatora ako su ispunjena sledeća dva uslova [36]:

1) Bazni klasifikatori treba da budu međusobno nezavisni. U tom slučaju bi, za problem binarne klasifikacije, konačna predikcija bila pogrešna samo ako više od polovine baznih klasifikatora pogreši u predikciji. Potpuna nezavisnost klasifikatora je teško ostvariva, ali se u praksi pokazalo da nije neophodna da bi se ostvarili bolji rezultati.

2) Bazni klasifikatori treba da budu bolji od slučajnog klasifikatora.

Osnovna ideja ansambl metoda je kreiranje velikog broja klasifikatora na osnovu trening podataka i neki vid agregacije njihovih rezultata u slučaju nepoznate instance. Ansambl klasifikatora može se konstruisati na neki od sledećih načina [36]:

Manipulacijom trening podataka Ponovnim izborom, u skladu sa izabranom raspodelom, iz originalnog skupa trening podataka kreira se veći broj novih

skupova podataka. Raspodela utiče na verovatnoću da će podatak biti izabran za novi trening skup. Zatim se za svaki novi trening skup kreira model klasifikacije izabranim algoritmom (npr. stablo odlučivanja). Dva ansambl metoda koja rade na ovaj način su upakivanje i podsticanje.

Upakivanje (eng. *bagging*) je tehnika koja iznova vrši izbor uzoraka sa zamenom na osnovu uniformne raspodele verovatnoća. Kako sve instance imaju jednaku verovatnoću da budu izabrane, ova tehnika je manje podložna greškama usled preprilagođavanja. Dobijeni skup uzoraka je iste veličine kao i originalni skup. Kako se izbor uzoraka vrši sa zamenom neki podaci će biti izabrani više puta a neki nijednom. Na dobijenim skupovima se treniraju klasifikatori. Nakon što se kreiraju svi klasifikatori moguće je klasifikovati nepoznate instance. Dodeljivanje klase nepoznatoj instanci vrši se tako što svaki bazni klasifikator dâ svoju predikciju klase, nakon čega se novoj instanci pridružuje najzastupljenija klasa.

Podsticanje (eng. *boosting*) je iterativna procedura koja postepeno menja raspodelu trening podataka i na taj način favorizuje one podatke koji su teži za klasifikaciju. Svakoj trening instanci je pridružen koeficijent težine koji se može promeniti po završetku iteracije. Težine se mogu koristiti bilo kao raspodela pri kreiranju novih skupova podataka bilo za treniranje pristrasnih modela.

Manipulacijom ulaznih atributa Novi skupovi trening podataka nastaju kao podskupovi originalnih podataka, slučajnim izborom podataka početnog skupa ili analizom domena. Ovaj pristup daje dobre rezultate u slučaju redundantnih atributa. Slučajne šume su primer metoda koji manipuliše ulaznim atributima i koristi klasifikatore zasnovane na stablima odlučivanja u svojoj osnovi.

Manipulacijom algoritma učenja Moguće je menjati i sam algoritam učenja. Ovaj način nalazi praktičnu primenu ubacivanjem faktora slučajnosti prilikom treniranja skupa stabala odlučivanja. Umesto izbora najboljeg atributa za podelu u čvoru, moguće je slučajnim izborom odabrati jedan od nekoliko najboljih atributa podele.

Manipulacijom oznaka klasa Ako je broj klasa dovoljno veliki, višeklasni problem je moguće transformisati u binarni problem podelom oznaka klasa, na slučajan način, u dva disjunktna skupa A_0 i A_1. Trening podacima čije klase

pripadaju skupu A_0 pridružuje se klasa 0, a podacima čije klase pripadaju skupu A_1 klasa 1. Podaci sa novim oznakama klasa se koriste za treniranje baznog klasifikatora. Ansambl klasifikatora se dobija višestrukim ponavljanjem kreiranja skupova A_0 i A_1 i treniranja klasifikatora nad dobijenim skupovima. Dodeljivanje klase nepoznatoj instanci vrši se tako što svaki bazni klasifikator C_i vrši predviđanje. Ako je predviđena klasa 0, sve klase skupa A_0 dobijaju glas. Analogno, ako je predviđena klasa 1, sve klase skupa A_1 dobijaju glas. Glasovi se prebrojavaju i rezultujuća klasa će biti ona sa najviše glasova.

Slučajne šume (eng. *Random forests*) pripadaju klasi ansambl metoda. Kao bazni klasifikatori se koriste stabla odlučivanja koja se konstruišu nad skupom nezavisnih slučajnih vektora koji nastaju metodom manipulacije trening podataka. *Upakivanje sa korišćenjem stabala odlučivanja* (eng. *Bagging using decision trees*) je specijalan slučaj slučajnih šuma u kom se slučajnost dodaje u proces pravljenja modela (do sada je postojala u trening podacima) tako što se novi skupovi podataka kreiraju metodom slučajnog izbora sa zamenom od elemenata polaznog skupa. Dodavanje slučajnosti smanjuje korelaciju između stabala, a samim tim i grešku generalizacije ansambla. Upakivanje popravlja grešku generalizacije smanjujuću varijansu baznih klasifikatora. Pored toga, upakivanje je manje podložno i greškama usled preprilagođavanja, koje je velika mana stabala odlučivanja, zato što nijedna trening instanca nema prednost prilikom izbora, već sve imaju podjednaku verovatnoću da će biti izabrane [36].

Svako stablo odlučivanja za podelu u čvoru koristi odgovarajući slučajni vektor podataka koji se dobija na jedan od sledećih načina [36]:

1) Slučajnim izborom se određuje F atributa koji će se koristiti za podelu u svakom od čvorova (umesto da se za podelu razmatraju svi atributi) nakon čega se stablo konstruiše do kraja bez odsecanja. Ako postoji potreba za većim faktorom slučajnosti moguće je koristiti upakivanje za kreiranje trening skupova. Izbor broja atributa F utiče na snagu i korelaciju modela slučajnih šuma. Za male vrednosti F stabla šume su slabo korelisana, dok veliko F omogućava jače modele zbog korišćenja većeg broja atributa. Često se kao kompromisno rešenje za broj atributa uzima $F = log_2 d + 1$, gde je d broj ulaznih atributa. Ovaj pristup vodi manjem vremenu izvršavanja algoritma jer se prililkom podele u čvoru ne

razmatraju svi atributi.

2) Ako početnih atributa d ima malo, teško je pronaći nezavisni skup slučajnih atributa koji se koriste pri izgradnji modela. Ovo se može prevazići proširenjem skupa atributa njihovom linearnom kombinacijom. Na nivou svakog čvora novi atribut nastaje tako što se metodom slučajnog izbora izabere L atributa iz polaznog skupa, nakon čega se ti atributi linearno kombinuju sa koeficijentima dobijenim iz uniformne raspodele na intervalu [-1,1]. Svaki čvor će dobiti F novih atributa, a najbolji od njih će biti korišćen za podelu čvora.

3) Mogući pristup podeli unutar čvora je da se umesto najboljeg atributa podele na slučajan način izabere jedan od F najboljih atributa. Stabla dobijena na ovaj način mogu imati veći stepen korelacije u slučaju nedovoljno velikog parametra F. Pored toga, ovaj pristup ne daje uštedu u vremenu izvršavanja kao prethodna dva pristupa jer je potrebno ispitati svaki atribut podele u svakom čvoru stabla.

3.2 Pretprocesiranje podataka

Pretprocesiranje podataka (eng. *data preprocessing*) je široka oblast koja obuhvata različite tehnike i strategije koje se koriste zarad dobijanja podataka pogodnijih za rad. Te tehnike i strategije spadaju ili u izbor objekata i atributa za analizu ili u kreiranje novih atributa/menjanje postojećih atributa i ne postoji generalno pravilo kojim redom će se one primenjivati. Na izbor tehnika i strategija utiče priroda podataka sa kojima se radi [36].

3.2.1 Transformacija atributa

Termin *transformacija atributa* označava transformaciju koja se primenjuje na sve vrednosti atributa. To znači da će vrednost atributa svakog objekta biti transformisana. Postoje različiti tipovi transformacija, a koji tip će biti korišćen zavisi od prirode podataka. U nekim situacijama je dovoljno primeniti jednostavnu matematičku funkciju na vrednosti atributa x (npr. x^k, $\log x$, $\sqrt{x}$, $|x|$, ...) . U nekim drugim situacijama potrebno je izvršiti normalizaciju ili standardizaciju [1]. U tezi se za transformaciju podataka koristi samo standardizacija pa će ona biti detaljno ob-

[1]U nekim knjigama ne postoji razlika između ova dva termina. Međutim, paket *scikit-learn* razlikuje pojmove normalizacije i standardizacije pa će stoga biti razlikovani i u ovoj tezi.

jašnjena. Cilj normalizacije i standardizacije je da transformacijama skup vrednosti dobije određena svojstva [36].

Standardizacija je tehnika transformisanja atributa kojom se svaki atribut transformiše oduzimanjem njegove srednje vrednosti i zatim deljenjem standardnom devijacijom. Na ovaj način se dobijaju nove vrednosti atributa koje su centrirane oko nule, a standardna devijacija će biti 1. Standardizacija je često neophodna za veliki broj algoritama mašinskog učenja paketa *scikit-learn*. U slučaju da atributi nemaju raspodelu podataka sličnu normalnoj ili Gausovoj algoritmi mašinskog učenja mogu proizvesti pogrešne rezultate. Standardizacija je takođe često neophodna ako se veći broj promenljivih međusobno kombinuju. Standardizacijom se sprečava uticaj promenljive sa visokim vrednostima atributa na rezultat izračunavanja. Jednostavan ali slikovit primer kada treba primeniti standardizaciju je situacija u kojoj treba porediti dve osobe na osnovu godina i plate. Razlike u platama mogu biti veće od razlika u godinama zato što se njihovi opsezi vrednosti dosta razlikuju (godine su broj manji od 100, a plate se mere u desetinama ili stotinama hiljada). U takvoj situaciji, ukoliko se ove razlike ne uzmu u obzir, na rezultat poređenja će najviše uticati razlike u platama (Euklidsko rastojanje je jedna od metrika kod koje se moraju uzeti u obzir razlike u opsezima atributa) [7, 21, 36].

3.2.2 Uzorkovanje

Jedan od načina rešavanja problema neizbalansiranih klasa je *uzorkovanje* (eng. *sampling*) sa ciljem modifikovanja raspodele instanci. Posledica ovakve modifikacije raspodele je da ređe zastupljena klasa postaje reprezentativna u skupu trening podataka tj. da se poveća njen uticaj na klasifikacioni model. Uzorkovanje obuhvata dodavanje novih instanci (eng. *oversampling*), uklanjanje postojećih instanci (eng. *undersampling*) i kombinaciju prethodne dve tehnike [36, 17]. Uzorkovanje je zapravo mnogo više od navedenog, ali detalji o uzorkovanju neće biti detaljnije opisani jer je fokus rada na dodavanju novih instanci.

Uklanjanje postojećih instanci je tehnika transformisanja trening skupa uklanjanjem nekih od podataka koji pripadaju zastupljenijoj klasi originalnog skupa podataka za trening. *Dodavanje novih instanci* je tehnika transformisanja trening skupa dodavanjem novih podataka nastalih na osnovu nekih od podataka koji pripadaju manje zastupljenoj klasi originalnog skupa podataka za trening [36].

3.2.3 Izbor atributa

Skupovi podataka mogu da sadrže podatke sa velikim brojem atributa tj. podatke koji imaju visoku dimenzionalnost. Jedan takav primer je *matrica terma dokumenata* (eng. *document term matrix*) čiji su redovi dokumenti, kolone su termi odnosno reči, a u preseku redova i kolona se nalaze vrednosti koje predstavljaju broj pojavljivanja reči u dokumentu. Ove matrice potencijalno mogu imati hiljade ili desetine hiljada atributa [36].

Postoje brojne prednosti smanjenja dimenzionalnosti: algoritam mašinskog učenja zahteva manje vremena i memorije, algoritam mašinskog učenja može pružati bolje performanse u slučaju manjih dimenzija jer se smanjenjem dimenzije eliminišu nebitni atributi i umanjuje šum, a takođe se i rešava problem prokletstva dimenzionalnosti [2], dobijeni model je lakši za tumačenje ako je manji broj atributa uključen i vizualizacija podataka je olakšana ako postoji manji broj atributa [36, 34].

Smanjenje dimenzionalnosti često se odnosi na tehnike koje do smanjenja vode pravljenjem novih atributa kombinovanjem postojećih. Do smanjenja dimenzionalnosti može se doći i tako što novi atributi predstavljaju podskup postojećih. Ova tehnika zove se *izbor podskupa atributa* (eng. *feature subset selection*) ili kraće *izbor atributa* (eng. *feature selection*). Iako deluje da bi se izborom atributa mogle izgubiti bitne informacije ovo nije slučaj ukoliko postoje nevažni i redundantni atributi. *Redundantni atributi* dupliraju informaciju koja je već postojana u nekom drugom atributu (npr. cena atrikla i porez), dok *nevažni atributi* ne sadrže informacije od značaja (npr. indeksi studenata prilikom predviđanja ocene na ispitu) [36].

Idealan pristup izboru atributa je isprobavanje svih mogućih kombinacija atributa za ulaz u algoritam mašinskog učenja. U praksi ovo nije izvodljivo jer je broj podskupova za n atributa jednak 2^n. Postoje tri često korišćene grupe pristupa za izbor atributa: 1) *ugrađeni pristupi* (eng. *embedded approaches*) - izbor atributa dešava se prilikom izvršavanja algoritma mašinskog učenja tj. prilikom obučavanja klasifikacionog modela kada algoritam bira koje atribute da zadrži a koje da zanemari, 2) *pristupi koji koriste filtere* (eng. *filter approaches*) - izbor atributa vrši se pre pokretanja algoritma mašinskog učenja i nezavisno od samog zadatka istraživanja podataka i 3) *pristupi koji koriste omotače* (eng. *wrapper approaches*) - za izbor

[2] *Prokletstvo dimenzionalnosti* (eng. *the curse of dimensionality*) je situacija u kojoj analiza podataka postaje sve teža sa porastom dimenzionalnosti.

atributa se, kao crna kutija, koristi neki algoritam mašinskog učenja [36, 34, 17].

3.3 Matrica konfuzije

Matrica konfuzije je tabelarni prikaz brojeva ispravno i pogrešno klasifikovanih objekata na osnovu kojih se mogu vršiti ocene modela klasifikacije. Slika 3.3 prikazuje matricu konfuzije za problem binarne klasifikacije. Svako od polja f_{ij} označava broj objekata klase i koji su klasifikovani kao objekti klase j. Za problem binarne klasifikacije sa nejednako zastupljenim klasama, ređe zastupljena klasa često nosi naziv pozitivna klasa, a zastupljenija klasa nosi naziv negativna klasa. Vrednosti matrice imaju sledeće značenje [36]:

Stvarno pozitivni (TP - true positive) ili f_{11} je broj objekata koji pripadaju pozitivnoj klasi a dodeljena im je pozitivna klasa.

Lažno negativni (FN - false negative) ili f_{10} je broj objekata koji pripadaju pozitivnoj klasi a dodeljena im je negativna klasa.

Lažno pozitivni (FP - false positive) ili f_{01} je broj objekata koji pripadaju negativnoj klasi a dodeljena im je pozitivna klasa.

Stvarno negativni (TN - true negative) ili f_{00} je broj objekata koji pripadaju negativnoj klasi a dodeljena im je negativna klasa.

		Predviđena klasa	
		Klasa $= 1$	Klasa $= 0$
Stvarna klasa	Klasa $= 1$	f_{11} (TP)	f_{10} (FN)
	Klasa $= 0$	f_{01} (FP)	f_{00} (TN)

Slika 3.3: Matrica konfuzije za problem binarne klasifikacije

Na osnovu vrednosti matrice mogu se izračunati ukupan broj tačnih predviđanja ($f_{11} + f_{00}$) i ukupan broj pogrešnih predviđanja ($f_{10} + f_{01}$). Za poređenje većeg broja modela bilo bi jednostavnije da su vrednosti matrice konfuzije predstavljene jednim brojem. U tu svrhu se mogu koristiti *tačnost* (eng. *accuracy*) i *odnos greške* (eng. *error rate*) [36]. Tačnost je broj tačnih predviđanja podeljen ukupnim brojem

predviđanja i za binarnu klasifikaciju definiše se formulom:

$$Tačnost = \frac{f_{11} + f_{00}}{f_{11} + f_{10} + f_{01} + f_{00}}$$

Odnos greške je broj pogrešnih predviđanja podeljen ukupnim brojem predviđanja i za binarnu klasifikaciju definiše se formulom:

$$Odnos\ greške = \frac{f_{10} + f_{01}}{f_{11} + f_{10} + f_{01} + f_{00}}$$

Većina algoritama treba da ostvari najveću tačnost (ili analogno najmanji odnos greške). U situacijama kada su klase nejednako zastupljene tačnost nije mera koja odgovara podacima jer svaku klasu tretira kao jednako važnu.

Metrike koje se sa dosta uspeha koriste kada je potrebno otkriti jednu od klasa su *preciznost* i *odziv* [36]. Preciznost (eng. *precision*) je procenat stvarno pozitivnih objekata među objektima koji su klasifikovani kao pozitivni i za binarnu klasifikaciju definiše se formulom:

$$Preciznost = \frac{TP}{TP + FP} = \frac{f_{11}}{f_{11} + f_{01}}$$

Veća preciznost takođe znači da postoji manji broj lažno pozitivnih objekata. Odziv (eng. *recall*) predstavlja procenat pozitivnih objekata koji su ispravno klasifikovani i za binarnu klasifikaciju definiše se formulom:

$$Odziv = \frac{TP}{TP + FN} = \frac{f_{11}}{f_{11} + f_{10}}$$

Visok odziv znači da postoji mali broj lažno negativnih objekata. Preciznost i odziv se mogu sumirati u metriku koja se zove F_1 *mera* (eng. F_1 *measure*) i za binarnu klasifikaciju definiše se formulom:

$$F_1\ mera = \frac{2 \cdot TP}{2 \cdot TP + FP + FN} = \frac{2 \cdot f_{11}}{2 \cdot f_{11} + f_{01} + f_{10}}$$

3.4 Tehnike evaluacije klasifikacionih modela

Po završetku kreiranja klasifikacionog modela korisno je testirati njegove performanse na skupu podataka koji mu je nepoznat, pri čemu je neophodno da taj skup podataka sadrži informacije o klasama. Ovakav vid testiranja predstavlja nepristrasnu ocenu generalizacije. Često korišćeni metodi evaluacije su [36, 26, 17, 34]:

Evaluacija pomoću skupa za testiranje (eng. *holdout method*) je tehnika koja originalni skup podataka deli na dva disjunktna podskupa, za obučavanje i za testiranje klasifikatora (npr. u odnosima 70:30, 60:40). Zatim se klasifikacioni model dobije na osnovu podataka za treniranje, nakon čega se performanse modela ocenjuju na podacima za testiranje. Na ovaj način se na osnovu podataka za testiranje može proceniti tačnost klasifikacije. Ovaj metod ima nekoliko nedostataka: 1) manji broj podataka za obučavanje modela (jer je deo podataka izdvojen za kasnije testiranje modela) zbog čega je moguće dobiti lošiji model nego kada se koriste svi podaci, 2) model može dosta zavisiti od izgleda skupova za treniranje i testiranje i 3) skupovi za treniranje i za testiranje sada zavise jedan od drugog - klasa koja je zastupljenija u jednom skupu automatski je manje zastupljena u drugom.

K-slojna unakrsna validacija (eng. *K-fold cross-validation*) je tehnika evaluacije klasifikacionih modela koja predstavlja bolji izbor u odnosu na evaluaciju pomoću skupa za testiranje. U opštem slučaju izvodi se tako što se originalni skup podataka deli na k jednakih podskupova (slojeva). Jedan podskup koristi se za testiranje, a svi ostali za treniranje. Dobijenim modelom se izvrše predviđanja na trenutnom sloju. Ovaj postupak se ponavlja u k iteracija tako da se svaki podskup tačno jednom koristi za testiranje. Po završetku svih iteracija potrebno je izračunati ocenu kvaliteta modela na osnovu svih prethodnih predviđanja.

3.5 Mašinsko učenje u Python-u

„Python je interpretirani, objektno-orijentisani viši programski jezik sa dinamičkom semantikom. Python-ova jednostavna sintaksa se lako uči i naglašava čitljivost koda koja za posledicu ima nisku cenu održavanja [14]". Python podržava veliki broj paketa i modula što vodi ponovnoj upotrebljivosti i modularnosti koda. Nepostojanje faze kompilacije ubrzava čitav proces pisanja i testiranja koda. Debagovanje je prilično jednostavno, greške proizvode izuzetke i ako se oni ne uhvate interpreter ispisuje *stanje steka* (eng. *stack trace*). Štaviše, često je najjednostavniji način debagovanja ubacivanje *print* naredbi [14].

Autor Pythona je Holanđanin Gido Van Rosum (hol. Guido Van Rossum). Ideju o Python-u dobija krajem 1980-ih kada je radio kao programer na Državnom insti-

tutu za matematiku i informatiku CWI (Centrum voor Wiskunde en Informatica) u jeziku ABC kojim je inspirisan. Implementaciju započinje decembra 1989-te, a prva verzija jezika (verzija 0.9.0) puštena je 1991-ve. Januara 1994-te izlazi verzija 1.0 koja se razvija sve do septembra 2000-te. Unapređena verzija 2.0 (poznata i kao Python 2) izlazi u oktobru 2000-te, a verzija 3.0 (Python 3) u decembru 2008-me. Zanimljivo je da razvoj verzije 2.0 nije zaustavljen izlaskom nove velike verzije 3.0. Razlog za to je činjenica da je Python 3 prva distribucija namerno napravljena da bude nekompatibilna sa prethodnim distribucijama. Poslednja i konačna verzija vezana za Python 2 je 2.7 iz jula 2010-te, koja je i dalje podržana (poslednja promena desila se u aprilu 2018-te (2.7.15)). Verzija 3.7.0 je najnovijeg datuma (jun 2018-te). Izmene koje Python 3 donosi su brojne, a neke od njih su: *print* više nije naredba već postaje funkcija, interfejsi poput rečnika i mapa vraćaju poglede i iteratore umesto listi, pojednostavljena su pravila operatora poređenja, napravljene su promene u samoj sintaksi,... [1, 9, 15].

3.5.1 Ekosistem SciPy

SciPy je ekosistem slobodnog softvera za matematiku, nauku i inženjerstvo. Njegovu osnovu čini programski jezik Python i grupa paketa od kojih su za mašinsko učenje najznačajniji [11]:

1) **NumPy** je osnovni paket za naučna izračunavanja u Python-u. Ovaj paket definiše numeričke nizove, matrične tipove, kao i osnovne operacije nad njima. Glavni objekat paketa je homogeni višedimenzioni niz koji je predstavljen kao tabela elemenata (najčešće brojeva), pri čemu su svi istog tipa i indeksirani torkom pozitivnih celih brojeva. Dimenzije niza se nazivaju *osama* (eng. *axis*). Klasa nizova *NumPy* paketa se naziva *ndarray*.

   - Tačka prostora npr. [1, 2, 1] sa svoje tri koordinate ima jednu osu (dimenziju). Osa se sastoji od 3 elementa pa se kaže da je dužine 3.
   - Dvodimenzioni niz, npr.

     $$[[1., 0., 0.],$$
     $$[0., 1., 2.]]$$

     ima dve ose. Prva osa ima 2 elementa, a druga 3 [11, 12].

2) **Pandas** je paket koji obezbeđuje strukture podataka za jednostavan i intuitivan rad sa podacima. Predstavlja nadgradnju *NumPy* biblioteke i napravljen je tako da se lako integriše u razna okruženja koja se koriste za naučna izračunavanja. Pogodan je za podatke koji su smešteni u tabelama, uređene i neuređene vremenske serije, proizvoljne matrice i sve oblike podataka dobijenih na osnovu statističkih posmatranja. *Pandas* uvodi dve strukture podataka, *Series* za jednodimenzione podatke i *DataFrame* za dvodimenzione [6].

3) **Matplotlib** je biblioteka koja služi za kreiranje različitih vrsta dijagrama. Većinu dijagrama moguće je kreirati u svega nekoliko linija koda, a sami dijagrami se mogu prilagoditi potrebama korisnika izborom stilova linija (pune, isprekidane,...), svojstava fontova, svojstava koordinatnih osa, itd [4].

Scikit-learn je modul koji implementira veliki broj algoritama mašinskog učenja. Pored algoritama za rešavanje problema klasifikacije, regresije i klasterovanja ovaj modul podržava i razne tehnike preprocesiranja podataka, kao i tehnike ocenjivanja modela. Zasniva se na bibliotekama *NumPy*, *SciPy* i *matplotlib* [10].

Glava 4

Implementacija

Postupak automatizovane verifikacije softvera zahteva veliku količinu vremena i resursa. U želji da se u ovom procesu uštedi na resursima i vremenu izvršavanja korišćena je binarna klasifikacija. Pritom rezultati binarne klasifikacije treba dovoljno tačno da predviđaju da li programi sadrže greške.

Na osnovu korpusa podataka koji sadrži atribute i informaciju o prisustvu greške u programskim kodovima kreirana su tri klasifikaciona modela, svaki zasnovan na različitom algoritmu. Od dobijenih modela izabran je model koji najviše odgovara podacima tj. model koji daje najtačnije rezultate. Zatim je implementiran program koji na osnovu programskog koda napisanog u C-u izračunava atribute koda korišćene u fazi obučavanja modela. Na taj način se dobijeni prediktivni model može primeniti na proizvoljne programe. Izračunavanje atributa proizvoljnih programa je omogućilo da se model evaluira na specifičnim programima korišćenim na takmičenju iz verifikacije (za ove programe su bili dostupni njihovi kodovi i informacija o prisustvu greške, ali ne i atributi).

4.1 Korišćeni podaci

Kao skup podataka za obučavanje modela klasifikacije korišćen je korpus podataka *jm1.arff* koji se nalazi na repozitorijumu *PROMISE* (eng. *PROMISE Software Engineering Repository*). Repozitorijum sadrži skupove podataka relevantne za istraživanja u oblasti softverskog inženjerstva. Najveći deo skupova podataka koristi se za predviđanje defekata u softveru. U tom slučaju korišćeni podaci su programi

predstavljeni atributima i njihovim vrednostima. Pored ovih podataka na repozitorijumu se nalaze i podaci sa metrikama relevantnim za predviđanje cene softvera (eng. *software cost estimation*), predviđanje uspešnosti ponovne upotrebljivosti softvera (eng. *predicting successful reuse*), praćenje zahteva (eng. *requirements tracing*), kao i transakcioni podaci (eng. *transaction data*) i podaci za neka numerička izračunavanja (eng. *data for numerical computation library*) [32].

Datoteka *jm1* predstavlja NASA-in softver, napisan u programskom jeziku C, koji se koristi u okviru sklapanja, transporta i lansiranja svemirskih raketa i letelica. Objekti u datoteci odgovaraju funkcijama pomenutog softvera. Funkcije imaju između 1 i 3442 linije koda, srednja vrednost broja linija iznosi 42,03 a medijana 23. Ove funkcije su opisane Halstedovim metrikama (eng. *Halstead metrics*) zasnovanim na brojevima operatora i operanada, kao i metrikama zasnovanim na brojevima linija koda. Halsted je tvrdio da je veća verovatnoća da kôd težak za čitanje sadrži grešku, pri čemu se složenost čitanja koda procenjuje prebrojavanjem koncepata funkcije (na primer, prebrojavanjem jedinstvenih operatora). Datoteka *jm1* sadrži i informacije o postojanju defekta u funkcijama (detalji o tome šta spada pod pojam defekta nisu dati) [3, 5, 28].

Datoteka se sastoji od 10878 objekata, od kojih svaki ima 22 atributa. Za potrebe ovog rada korišćeno je 18 atributa. Izostavljeni su atributi *cyclomatic complexity*, *design complexity*, *essential complexity* i *branch count*. Ovi atributi su izostavljeni jer za njih nije pronađen jedinstven opis, a izborom atributa je utvrđeno da ne spadaju u 10 najbitnijih atributa. Atributi su izostavljani tek nakon što je utvrđeno da njihovo izostavljanje neće značajno uticati na kvalitet modela klasifikacije - model dobijen bez ovih atributa ima preciznost svega 0,02 manju od modela koji koristi sve atribute. Zbog toga je procenjeno da je bolje poboljšavati model na druge načine. Sedamnaest atributa tipa *numeric* se odnosi na razne karakteristike koda, a osamnaesti je atribut klase (da li program ima ili nema grešku) [3, 20]:

LOC_BLANK Broj praznih linija u izvornom kodu programa

LOC_COMMENTS Broj linija komentara

LOC_TOTAL Ukupan broj linija koda

LOC_EXECUTABLE Broj linija koda koji se dobije kada se od ukupnog

broja linija oduzmu brojevi praznih linija i linija sa komentarima

LOC_CODE_AND_COMMENT Broj linija koda koji se dobije kada se od ukupnog broja linija oduzmu brojevi praznih linija

NUM_OPERATORS N_1 Ukupan broj operatora u programu

NUM_OPERANDS N_2 Ukupan broj operanada u programu

NUM_UNIQUE_OPERATORS η_1 Broj jedinstvenih operatora u programu

NUM_UNIQUE_OPERANDS η_2 Broj jedinstvenih operanada u programu

HALSTEAD_LEVEL $L = (2 \cdot \eta_2)/(\eta_1 \cdot N_2)$

HALSTEAD_VOLUME $V = (N_1 + N_2) \cdot (log_2(\eta_1 + \eta_2))$

HALSTEAD_LENGTH $N = N_1 + N_2$

HALSTEAD_CONTENT $I = L \cdot V$

HALSTEAD_DIFFICULTY $D = 1/L = (\eta_1/2) \cdot (N_2/\eta_2))$

HALSTEAD_EFFORT $E = V/L = D \cdot V$

HALSTEAD_ERROR_EST $B = V/3000$

HALSTEAD_PROG_TIME $T = E/18$

Defective {Y,N} Atribut klase sa vrednostima Y (program ima grešku) i N (program nema grešku)

4.1.1 Format arff

Format *arff* (eng. *Attribute-Relation File Format*) predstavlja način reprezentacije skupova podataka koji se sastoje od nezavisnih, neuređenih instanci i koji ne uključuju međusobne relacije među instancama. Izgled *arff* fajla prikazan je na slici 4.1. Linije koje počinju znakom „%” predstavljaju komentare. Nakon komentara nalaze se naziv relacije i blok koji definiše atribute. U primeru sa slike prikazani su imenski i brojčani atributi. U okviru formata *arff* takođe su podržani i atributi

čiji je tip string, kao i atributi čiji je tip datum. Pri definiciji imenskih atributa potrebno je osim naziva atributa navesti i skup vrednosti, u vitičastim zagradama, koje atribut uzima. Brojčani atributi moraju da sadrže ključnu reč *numeric*, stringovi *string*, a datumi *date*. Tip datum je zapravo poseban slučaj tipa string koji poštuje datum-vreme format: *yyyy-MM-dd-THH:mm:ss* (četiri cifre za godinu, po dve za mesec i dan nakon čega sledi karakter *T* i po dve cifre za sate, minute i sekunde). Odlika *arff* formata je da se klasni atribut eksplicitno ne navodi već je on jedan od navedenih atributa. Posledica ove odlike je da se isti fajl može koristiti da bi se odredilo koliko dobro se predviđa svaki atribut na osnovu ostalih, za pronalaženje pravila pridruživanja (eng. *association rules*) ili za klasterovanje. Nakon bloka kojim se definišu atributi slede linije sa podacima, kojima prethodi *@data* linija. Svaka instanca se navodi u posebnoj liniji, vrednosti atributa su razdvojene zarezima, a nedostajuće vrednosti označene znakom ? [37].

```
% arff fajl koji sadrzi podatke o vremenskim
% uslovima
%

@relation vreme

@attribute izgled {suncano, oblacno, kisovito}
@attribute temperatura numeric
@attribute vlaznost_vazduha numeric
@attribute duva_vetar {da, ne}
@attribute pogodno_za_sport {da, ne}

@data
suncano, 24, 60, ne, da
oblacno, 15, 55, ne, ne
kisovito, 22, 80, da, ne
suncano, 36, 86, ne, ne
oblacno, 27, 52, ne, da
kisovito, 22, ?, da, ne

```

Slika 4.1: Primer fajla formata *arff*

4.2 Nalaženje najboljeg klasifikatora

U cilju dobijanja najboljeg modela klasifikacije, konstruisani su klasifikatori zasnovani na algoritmima k najbližih suseda, logističke regresije i slučajnih šuma. Svaki od klasifikatora tretiran je na isti način. Vršene su razne transformacije da

bi dobijeni klasifikatori davali preciznije rezultate. Nakon izvršenih transformacija, upoređivani su dobijeni rezultati. Poređenje klasifikatora vršeno je upoređivanjem nekih od svojstava klasifikatora. U nastavku je opisan proces kreiranja i unapređivanja klasifikacionih modela.

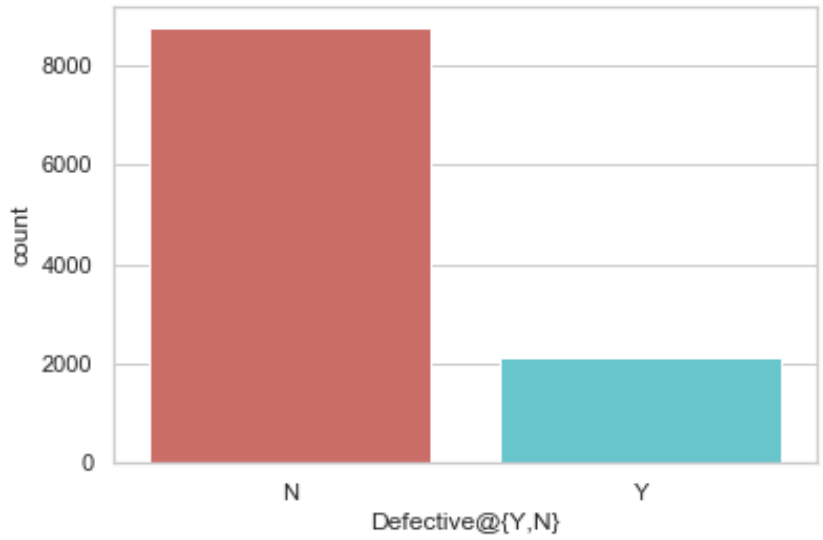
4.2.1 Upoznavanje sa podacima

Upoznavanje sa podacima je neizostavan korak u procesu dobijanja kvalitetnog klasifikacionog modela. Od prirode podataka zavisi koji algoritmi se mogu koristiti za dobijanje modela, kao i kakve transformacije podataka su potrebne ili neophodne da bi dobijeni modeli bili verodostojni. Zbog toga je najpre potrebno upoznati se sa raspoloživim podacima. Broj podataka i atributi su poznati iz dela 4.1. Jedna od prvih provera je da li postoje podaci sa nedostajućim vrednostima. Takvih podataka nema u korišćenom korpusu, tako da neće biti više reči na ovu temu. Takođe je potrebno proveriti raspodelu klasa među podacima. Na slici 4.2 vidi se da su podaci prilično neravnomerno raspoređeni. Podaci koji nemaju grešku su daleko zastupljeniji od podataka koji imaju grešku. Od 10878 podataka, čak 8776 podataka nema grešku, a preostalih 2102 ima grešku što može predstavljati problem. Ovakva pojava predstavlja problem neizbalansiranih klasa. Modeli trenirani nad podacima koji imaju ovako velike razlike u raspodeli klasa mogu davati pristrasne rezultate tj. težiti da novim instancama dodele zastupljeniju klasu (u ovom slučaju je to klasa programa koji nemaju greške). Kako su podaci od interesa programi kod kojih greška postoji takvo ponašanje modela nije poželjno. Problem neizbalansiranih klasa se može rešiti na nekoliko načina. U ovom radu problem je rešen dodavanjem novih instanci. Postoji nekoliko koraka urađenih pre dodavanja novih instanci kojima je potrebno posvetiti više pažnje.

4.2.2 Transformacija atributa

Od prirode raspoloživih podataka zavisi koja transformacija atributa će se primeniti. Zato je potrebno videti kako atributi izgledaju i koje vrednosti uzimaju.

Slika 4.3 prikazuje kako se opsezi vrednosti pet prikazanih atributa dosta razlikuju. Na primer, vrednosti atributa *HALSTEAD_LEVEL* su brojevi u intervalu [0, 1] i one su izrazito male ako se porede sa vrednostima drugih prikazanih atributa. Razlika je posebno primetna ako se posmatraju vrednosti atributa *HAL-*



Slika 4.2: Raspodela objekata prema atributu klase

HALSTEAD_LENGTH@NUMERIC	HALSTEAD_LEVEL@NUMERIC	HALSTEAD_PROG_TIME@NUMERIC	HALSTEAD_VOLUME@NUMERIC	NUM_OPERANDS@NUMERIC
2702.0	0.01	193552.92	19997.18	784.0
8441.0	0.00	1726654.57	80843.08	3021.0
0.0	0.00	0.00	0.00	0.0
4828.0	0.00	514156.64	43342.31	1730.0
685.0	0.02	12891.31	5009.32	295.0

Slika 4.3: Opsezi vrednosti za neke od atributa

STEAD_ PROG_ TIME (za drugi objekat sa slike vrednost ovog atributa iznosi oko 1,7 miliona!). Zbog toga je potrebno izvršiti standardizaciju podataka. U suprotnom, atributi sa visokim vrednostima bi previše uticali na rezultat klasifikacije.

4.2.3 Treniranje modela klasifikacije i njihove ocene

Nakon osnovnog upoznavanja sa podacima i urađene standardizacije mogu se konstruisati inicijalni modeli klasifikacije za svaki algoritam. Rad sa klasifikacionim modelima u Python-u je prilično jednostavan i gruba skica bi se sastojala od četiri koraka [10]

1. Na početku je potrebno skup podataka podeliti na skup za trening i skup za testiranje (npr. u odnosu 60:40) da bi kasnije bilo moguće proceniti kvalitet modela. Za podelu skupa podataka dovoljno je pozvati metod *train_ test_ split*

koji pripada modulu *model_selection* u okviru paketa *scikit-learn*. Treba napomenuti da je podela skupa podataka na skupove za treniranje i testiranje (kao i unakrsna validacija koja je takođe korišćena u tezi) samo način da se izvrši procena kvaliteta dobijenog modela (kao što je opisano u delu 3.4). Konačni model koji će se koristiti za dodeljivanje klasa nepoznatim podacima treba da se obuči nad celokupnim skupom raspoloživih podataka [26].

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test =
        train_test_split(X, y, test_size=0.4, random_state=42)
```

2. Zatim se iz odgovarajućeg paketa učitava i kreira klasifikacioni model. Što se tiče modela zasnovanih na algoritmu k najbližih suseda u ovoj tezi se za vrednosti parametra k uzimaju $k = 3$ i $k = 15$. Do konkretnih vrednosti se došlo konstruisanjem modela za vrednosti $k = 1..25$ i poređenjem tačnosti dobijenih modela. Modeli dobijeni za vrednosti $k = 3$ i $k = 15$ su modeli koji su imali najveću ili skoro najveću tačnost za neparnu vrednost parametra k. Za model zasnovan na algoritmu logističke regresije parametar *random_state* koji se koristi za generisanje pseudoslučajnih brojeva postavljen je na nulu. Pored toga vrednosti parametara *solver* i *max_iter* su promenjene tako da budu u skladu sa narednom verzijom biblioteke, čime su eliminisana upozorenja. Modelu zasnovanom na algoritmu slučajne šume je parametar *random_state* takođe postavljen na nulu, a parametar *n_estimators* je promenjen u skladu sa narednom verzijom biblioteke.

```
# from sklearn.neighbors import KNeighborsClassifier
# clf=KNeighborsClassifier(n_neighbors=15)
# from sklearn.linear_model import LogisticRegression
# clf=LogisticRegression(random_state=0,
#                        solver='lbfgs', max_iter=150)
from sklearn.ensemble import RandomForestClassifier
clf=RandomForestClassifier(random_state=0, n_estimators=100)
```

3. Klasifikator uči model na osnovu trening podataka pomoću metoda *fit*.

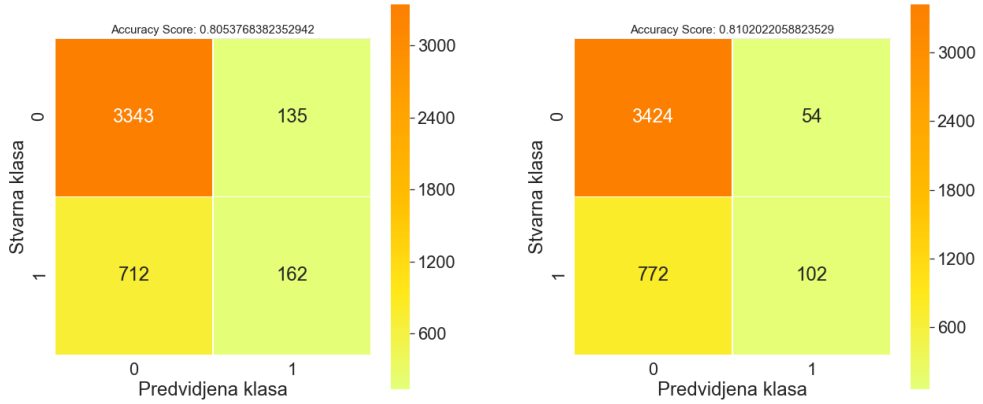
```
clf.fit(X_train, np.ravel(y_train))
```

4. Konačno, metodom *predict* se model primeni na test podatke.

```
y_pred = clf.predict(X_test)
```

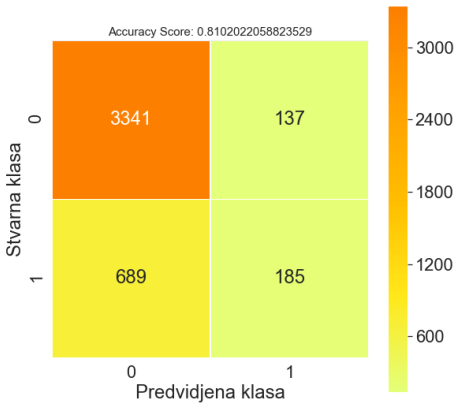
Na slici 4.4 mogu se videti tačnosti i matrice konfuzije za svaki od tri klasifikaciona modela. Sva tri algoritma daju modele sličnih tačnosti: 0,805, 0,810 i 0,810 (10-slojnom unakrsnom validacijom su dobijene tačnosti modela 0,775, 0,797 i 0,776). Međutim, imajući u vidu da zastupljenija klasa čini oko 81% podataka može se smatrati da ovi modeli ništa nisu naučili.

Sledeći korak u potrazi za boljim modelima je rešavanje problema neizbalansiranih klasa dodavanjem novih instanci.



(a) k najbližih suseda

(b) logistička regresija



(c) slučajne šume

Slika 4.4: Matrice konfuzije za klasifikatore zasnovane na različitim algoritmima

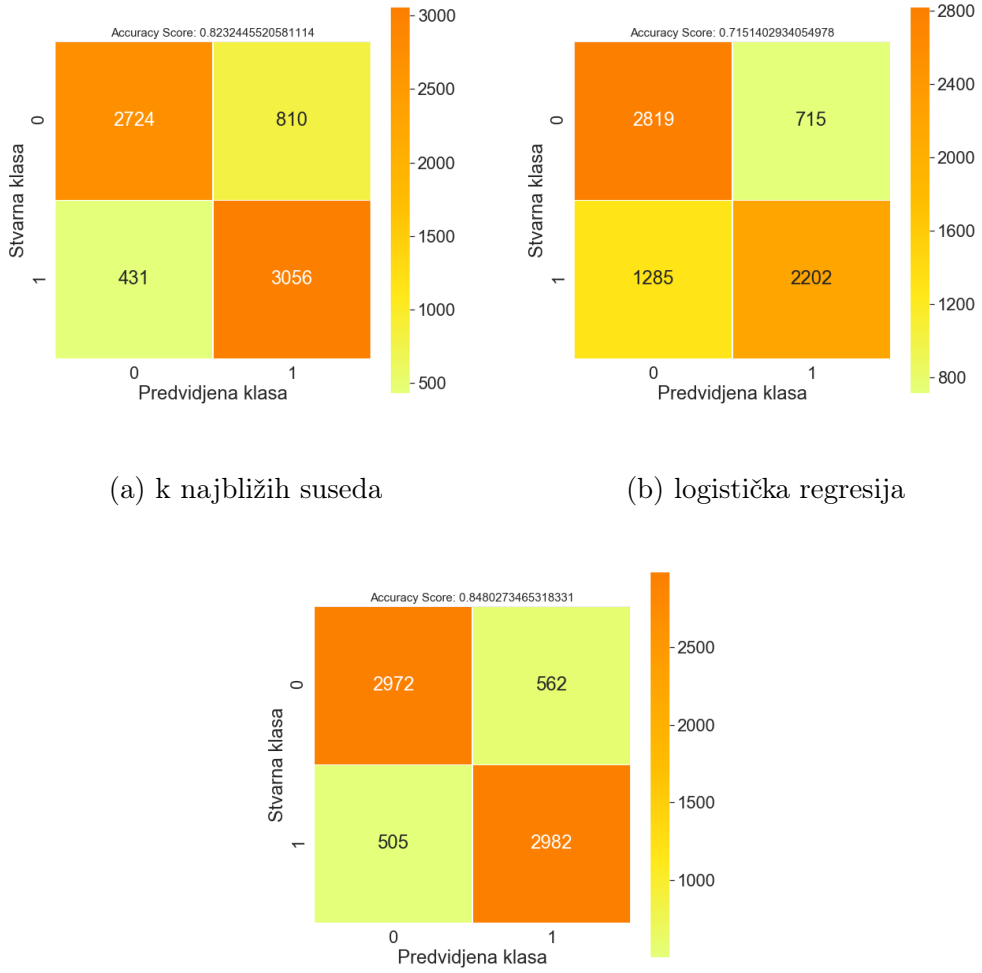
4.2.4 Dodavanje novih instanci - Oversampling

U Python-u je dodavanje novih instanci podržano u okviru paketa *imbalanced-learn* koji pruža mogućnost korišćenja četiri metoda dodavanja novih instanci - slučajno dodavanje (*RandomOverSampler*), *SMOTE*, *SMOTE-NC* i *ADASYN*. *RandomOverSampler* kreira nove uzorke na osnovu postojećih podataka metodom slučajnog izbora sa zamenom. *ADASYIN* i *SMOTE* nove uzorke kreiraju interpolacijom. Fokus *ADASYIN-a* je na kreiranju podataka koji se nalaze u blizini originalnih podataka pogrešno klasifikovanih od strane algoritma k najbližih suseda. Sa druge strane, *SMOTE* podjednako tretira sve podatke (bez obzira na to kako su klasifikovani algoritmom k najbližih suseda). *SMOTE-NC* služi za rad sa skupovima podataka koji sadrže neprekidne i kategoričke atribute. U ovoj tezi korišćen je metod *SMOTE* [2].

```
from imblearn.over_sampling import SMOTE
X_resampled, y_resampled = SMOTE(kind="svm").fit_sample(X, y)
```

Nakon dodavanja novih podataka potrebno je ponovo proći postupak obučavanja modela. Odgovarajuće matrice konfuzije prikazane su na slici 4.5. Procenjene tačnosti modela na osnovu podele skupa podataka na trening i test skup iznose 0,823, 0,715 i 0,848, a na osnovu 10-slojne unakrsne validacije 0,832, 0,650 i 0,800. Ono što iskače u prvi plan je vidno lošija tačnost modela zasnovanog na algoritmu logističke regresije što se posebno može primetiti evaluacijom pomoću tehnike unakrsne validacije. Takođe, modeli zasnovani na algoritmima k najbližih suseda i slučajne šume deluju stabilnije prilikom obe vrste evaluacije.

Dodatne informacije o modelima prikazane su na slici 4.6. Ponovo je interesantna preciznost, ovaj put modela 4.6a i 4.6c. U slučaju klase 0 preciznost iznosi 0,86 za k najbližih suseda i 0,85 za slučajne šume, a u slučaju klase 1 bolje rezultate daje model zasnovan na algoritmu slučajne šume (0,87 naspram 0,79). Zbog toga što je klasa od interesa klasa 1 (programi koji sadrže grešku) trenutno najbolji model je model zasnovan na algoritmu slučajne šume. Osim toga, na osnovu dobijenih rezultata deluje da model zasnovan na algoritmu logističke regresije ne uspeva dobro da prikaže prirodu podataka. Još jednu transformaciju je potrebno uraditi pre konačne presude.



(a) k najbližih suseda

(b) logistička regresija

(c) slučajne šume

Slika 4.5: Matrice konfuzije za klasifikatore zasnovane na različitim algoritmima nakon dodavanja novih podataka

4.2.5 Izbor atributa

Scikit-learn pruža nekoliko načina za izbor atributa, a u ovoj tezi je korišćen izbor atributa koji se zasniva na *značaju atributa*. Model *ExtraTreesClassifier* konstuiše veći broj stabala odlučivanja na različitim podskupovima originalnog skupa podataka, a zatim pomoću tehnike uprosečavanja pokušava da poboljša tačnost predviđanja i da kontroliše preprilagođavanje. Po završetku ovog posla pomoću metoda *feature_importances_* dobijaju se vrednosti koje odgovaraju značaju atributa [10].

	precision	recall	f1-score	support
0	0.86	0.77	0.81	3534
1	0.79	0.88	0.83	3487
micro avg	0.82	0.82	0.82	7021
macro avg	0.83	0.82	0.82	7021
weighted avg	0.83	0.82	0.82	7021

(a) k najbližih suseda

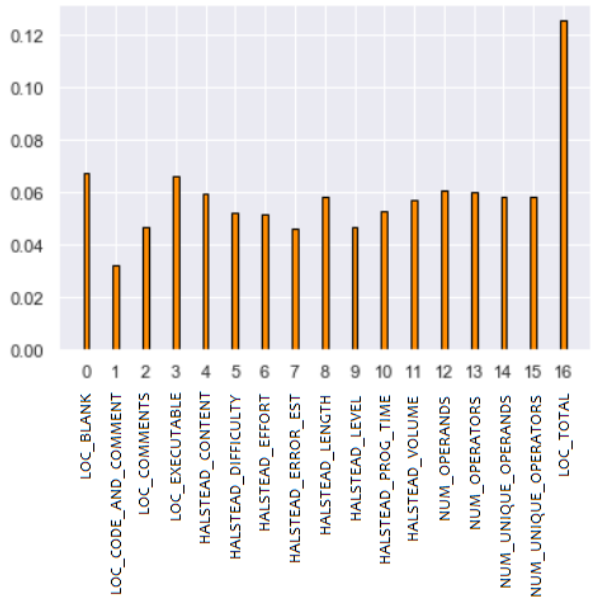
	precision	recall	f1-score	support
0	0.85	0.84	0.85	3534
1	0.84	0.86	0.85	3487
micro avg	0.85	0.85	0.85	7021
macro avg	0.85	0.85	0.85	7021
weighted avg	0.85	0.85	0.85	7021

(b) logistička regresija

	precision	recall	f1-score	support
0	0.85	0.87	0.86	3534
1	0.87	0.84	0.85	3487
micro avg	0.86	0.86	0.86	7021
macro avg	0.86	0.86	0.86	7021
weighted avg	0.86	0.86	0.86	7021

(c) slučajne šume

Slika 4.6: Preciznost, odziv, f1-mera i podrška za klasifikatore zasnovane na različitim algoritmima nakon dodavanja novih podataka



Slika 4.7: Značaj atributa

Na slici 4.7 prikazan je rezultat poziva metoda *feature_importances_*. Na x osi se nalaze indeksi atributa, a na y osi značaj atributa. U cilju poboljšanja tačnosti i/ili brzine modela na osnovu značaja atributa biće sačuvano 10 najznačajnijih atributa a ostali će biti odbačeni. Tih 10 najznačajnijih atributa, u rastućem poretku su: *HALSTEAD_VOLUME, HALSTEAD_LENGTH, NUM_UNIQUE_OPERATORS, NUM_UNIQUE_OPERANDS, HALSTEAD_CONTENT, NUM_OPERATORS, NUM_OPERANDS, LOC_EXECUTABLE, LOC_BLANK i LOC_TOTAL.*

4.2.6 Konačni izgled modela

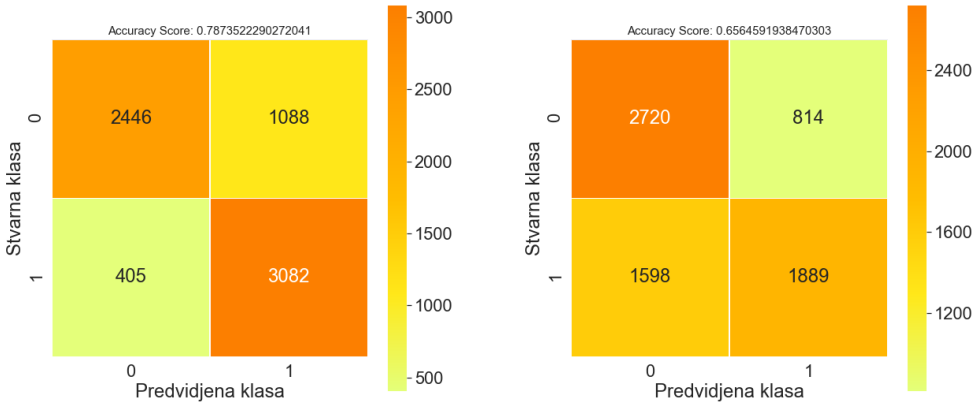
Slika 4.8 pokazuje da izborom atributa samo model zasnovan na algoritmu slučajne šume dobija na tačnosti, ali neznatno (0,854 u odnosu na 0,848). Lošije rezultate ostvaruju modeli zasnovani na algoritmima k najbližih suseda (0,787 u odnosu na 0,823) i logističke regresije (0,656 u odnosu na 0,715). To potvrđuju i rezultati 10-slojne unakrsne validacije 0,840, 0,564 i 0,780. Slika 4.9 pokazuje da su preciznosti sva tri modela nešto lošije nakon izbora atributa. Najveća razlika se vidi kod modela zasnovanog na algoritmu logističke regresije. Najmanju razliku u preciznosti (a takođe i u tačnosti modela) ima model zasnovan na algoritmu slučajne šume.

Na osnovu do sada viđenog može se zaključiti da je model koji najviše odgovara podacima model koji je zasnovan na algoritmu slučajnih šuma. Izborom atributa nisu dobijeni posebno bolji rezultati u smislu tačnosti modela, ali rezultati nisu ni mnogo lošiji u odnosu na model dobijen nad podacima kod kojih nije rađen izbor atributa. Zbog toga, u nadi da će to dovesti do uštede u vremenu obučavanja, prilikom obučavanja konačnog modela biće primenjen izbor atributa. Konačan model koji će se koristiti za klasifikaciju nepoznatih programa biće model zasnovan na algoritmu slučajnih šuma, obučavaće se nad svim raspoloživim podacima pri čemu će nad podacima biti urađen izbor atributa.

4.3 Program koji iz C koda dobija atribute

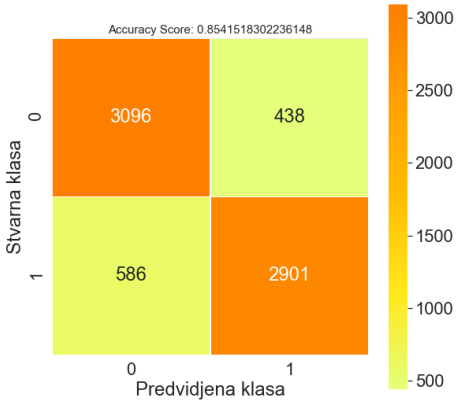
Ulazi modela klasifikacije su atributi programa napisanih u programskom jeziku C. Da bi se za proizvoljan program mogla raditi klasifikacija neophodno je njegov izvorni kôd prevesti u atribute. U tu svrhu implementiran je program koji na osnovu izvornog koda programa napisanog u C-u dobija atribute opisane u delu 4.1. Ovaj program napisan je u jeziku Python i većim delom se zasniva na biblioteci *pycparser*. *Pycparser* je parser programskog jezika C implementiran u Python-u. Detalji o modulu mogu se pronaći na linku: `https://github.com/eliben/pycparser` odakle se može izvršiti i njegovo preuzimanje. Korisne informacije se mogu naći i na veb strani autorke: `https://eli.thegreenplace.net/`.

Kako se atributi sa kojima se radi mogu podeliti u dve grupe (atributi koji se odnose na Halstedove metrike i atributi koji se odnose na linije koda) tako se



(a) k najbližih suseda

(b) logistička regresija



(c) slučajne šume

Slika 4.8: Matrice konfuzije za klasifikatore zasnovane na različitim algoritmima nakon dodavanja novih podataka i izbora atributa

postupak izračunavanja atributa na osnovu C koda može opisati u dva koraka.

Izračunavanje atributa koji se odnose na Halstedove metrike Halstedove metrike se koriste da bi se izmerili veličina softverskog proizvoda, kao i cena i napor potrebni za njegov razvoj. Ove metrike se zasnivaju na broju operatora i broju operanada izvornog koda (operatori i operandi o kojima je ovde reč nisu definisani na isti način kako se operatori i operandi definišu u programskom jeziku, već su to širi pojmovi koji obuhvataju i neke dodatke). Preciznije, u

	precision	recall	f1-score	support
0	0.86	0.69	0.77	3534
1	0.74	0.88	0.81	3487
micro avg	0.79	0.79	0.79	7021
macro avg	0.80	0.79	0.79	7021
weighted avg	0.80	0.79	0.79	7021

(a) k najbližih suseda

	precision	recall	f1-score	support
0	0.63	0.77	0.69	3534
1	0.70	0.54	0.61	3487
micro avg	0.66	0.66	0.66	7021
macro avg	0.66	0.66	0.65	7021
weighted avg	0.66	0.66	0.65	7021

(b) logistička regresija

	precision	recall	f1-score	support
0	0.84	0.88	0.86	3534
1	0.87	0.83	0.85	3487
micro avg	0.86	0.86	0.86	7021
macro avg	0.86	0.85	0.86	7021
weighted avg	0.86	0.86	0.86	7021

(c) slučajne šume

Slika 4.9: Preciznost, odziv, f1-mera i podrška za klasifikatore zasnovane na različitim algoritmima nakon dodavanja novih podataka i izbora atributa

pitanju su četiri atributa koda: broj jedinstvenih operatora, broj jedinstvenih operanada, ukupan broj operatora i ukupan broj operanada. Smernice za izračunavanje operanada i operatora za programski jezik C izgledaju ovako [30]:

Operandi

(a) Identifikatori - svi identifikatori koji ne spadaju u ključne reči

(b) Imena tipova - *int*, *float*, *char*, *double*, *long*, *short*, *signed*, *unsigned*, *void*, ...

(c) Konstante - bilo da im je sadržaj karakter, string ili broj

Operatori

(a) Specifikatori koji se odnose na skladišta - *auto*, *extern*, *register*, *static*, *typedef*

(b) Kvalifikatori tipova - *const*, *final*, *volatile*

(c) Razne rezervisane reči kao što su: *break*, *case*, *continue*, *default*, *do*, *if*, *else*, *enum*, *for*, *goto*, *if*, *new*, *return*, *sizeof*, *struct*, *switch*, *union*, *while*.

(d) Operatori - svi operatori programskog jezika C

Potrebno je napomenuti i neke specijalne slučajeve [30]:

1. Par zagrada je jedan operator
2. Jedan operator su takođe i delimiter ; kao i ternarni operator ?:

3. Labele su operatori ako se koriste u okviru *goto* konstrukta
4. Dvotačka u okviru *case:*, kao i par zagrada u okviru *for()*, *if()*, *while()*, *switch()* predstavljaju deo konstrukta i jedan operator
5. Komentari ne spadaju ni u operatore ni u operande
6. Imena funkcija su operatori ako je u pitanju poziv funkcije. Imena funkcija u deklaraciji i definiciji nisu operatori
7. Prethodno pravilo takođe važi i za promenljive i konstante

Pomoću biblioteke *pycparser* moguće je na osnovu datih smernica prebrojati operatore i operanade. Izračunavanje atributa koji se zasnivaju na operandima i operatorima je malo složenije od jednostavnog prebrojavanja i započinje parsiranjem C koda metodom *parse_file*. Rezultat parsiranja je *AST stablo*. AST stablo (eng. *Abstract Syntax Tree*) je drvolika reprezentacija sintakse izvornog koda programa pogodna za analizu i manipulaciju. Dobijeno AST stablo prepoznaje 47 tipova čvorova (neki od njih su: *Assignment*, *BinaryOp*, *For*, *If*, *Return*, *Struct*, *Switch*, *Typedef*, *While*, ...) koji se odnose na strukturu C koda. Obilaskom AST stabla, analizom njegovih čvorova i primenom prethodno navedenih smernica na čvorove stabla zabeleženi su svi operandi i većina operatora. Operatori koji su ostali nezabeleženi su: ( ) [ ] { } , ; : # ## ' i ". Ovi operatori su prebrojani regularnim izrazima zbog jednostavnosti. Po završetku prebrojavanja operatora i operanada moguće je izračunati atribute koji se na njih odnose. Nakon toga jednostavnom primenom formula iz 4.1 mogu se izračunati potrebne Halstedove metrike [8].

Izračunavanje atributa koji se odnose na linije koda Atributi koji se odnose na linije koda programa su računati korišćenjem regularnih izraza i prebrojavanjem linija koda. Ukupan broj linija koda *LOC_TOTAL*, kao i broj praznih linija *LOC_BLANC* dobijaju se prebrojavanjem. Broj linija komentara *LOC_COMMENTS* se računa korišćenjem regularnog izraza. Broj izvršivih linija koda *LOC_EXEC* se računa oduzimanjem broja praznih linija i broja linija sa komentarima od ukupnog broja linija. Broj linija koda i komentara *LOC_CODE_AND_COMMENT* se računa oduzimanjem broja praznih linija od ukupnog broja linija.

4.4 Primena klasifikacionog modela na izlaz programa za dobijanje atributa

Uz poznavanje najboljeg modela klasifikacije (modela zasnovanog na algoritmu slučajne šume) i uz program koji od izvornog koda programa napisanih u programskom jeziku C dobija atribute koji se koriste kao ulaz u klasifikacioni model, moguće je testirati ponašanje modela pri klasifikovanju programa koje model prvi put vidi. Korpus programa koji se koriste za testiranje modela je korpus korišćen na takmičenju u verifikaciji i preuzet je sa `https://github.com/sosy-lab/sv-benchmarks/tree/master/c`[1]

Imena korišćenih programa su oblika: ime programa, nakon čega sledi jedan ili više delova *_false-<svojstvo>* ili *_true-<svojstvo>* i .c ekstenzija. Na primer, program *cstrcat-alloca_true-valid-memsafety_true-termination.c* koji zadovoljava svojstva *valid-memsafety* i *termination*. Postoji nekoliko testiranih svojstava programa [13]:

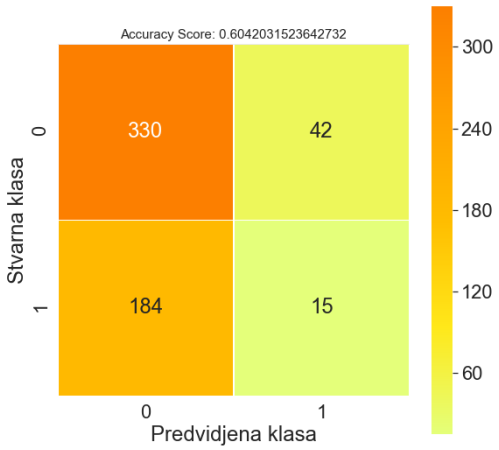
- *unreach-call*: Određeni poziv funkcije mora biti nedostižan u okviru programa.
- *valid-memsafety*, *valid-deref*, *valid-free*, *valid-memtrack*: Određeno svojstvo bezbednosti memorije mora biti zadovoljeno u programu, pri čemu je *memsafety* konjukncija preostala tri svojstva.
- *valid-memcleanup*: Sva alocirana memorija mora biti dealocirana pre nego što se program završi.
- *no-overflow*: Program ne sme sadržati nedefinisano ponašanje koje je posledica prekoračenja označenih brojeva.
- *termination*: Program se mora završiti na svakoj od izvršnih putanja.

Na osnovu prethodno definisanih svojstava programa poznato je koji programi sa takmičenja imaju grešku, a koji programi nemaju grešku (programi sa greškom imaju prefiks *false* ispred makar jednog od svojstava). Poznavanje klasa je neophodna informacija za testiranje performansi modela na novim podacima. Da bi

[1]Korpus korišćen u tezi je deo podskupa sa linka.

bilo moguće primeniti klasifikacioni model na korpus podataka sa takmičenja, potrebno je transformisati podatke na isti način kao i prilikom obučavanja modela - standardizacijom i izborom atributa.

Na slici 4.10 je prikazana matrica konfuzije dobijena primenom konačnog modela klasifikacije na programe sa takmičenja u verifikaciji. Ostvarena tačnost modela iznosi 0,604 što je oko 25% lošije u odnosu na rezultate dobijene nad podacima za testiranje. Od ukupno 571 programa sa takmičenja 372 nema grešku, a preostalih 199 ima. Pored pomenute tačnosti, dodatni problem predstavlja činjenica da je većini programa (čak 514) dodeljena klasa 0 - klasa programa bez greške. Od svih programa sa greškom samo 15 je ispravno klasifikovan. Tek 57 programa je označeno kao programi koji sadrže grešku. Nasuprot tome čak 184 programa koji zapravo sadrže grešku označeni su kao programi koji grešku ne sadrže. Kako su programi sa greškom programi od interesa, ovakvo ponašanje je problematično. Nešto detaljnije (ili bar razumljivije) informacije prikazane su na slici 4.11 gde se vidi da je i preciznost za obe klase (a posebno za klasu programa sa greškom) lošija u odnosu na preciznost modela primenjenog na podacima za testiranje. Odziv za klasu programa sa greškom iznosi svega 0,08 što je prilično loše.



Slika 4.10: Matrica konfuzije dobijena primenom modela zasnovanog na algoritmu slučajne šume na programima sa takmičenja

Jedna od ideja za ovaj rad bila je da se u slučaju da model dobro klasifikuje programe njegovi rezultati porede sa rezultatima nekog verifikacijskog alata. Poređenje

	precision	recall	f1-score	support
0	0.64	0.89	0.74	372
1	0.26	0.08	0.12	199
micro avg	0.60	0.60	0.60	571
macro avg	0.45	0.48	0.43	571
weighted avg	0.51	0.60	0.53	571

Slika 4.11: Preciznost, odziv, f1-mera i podrška za model zasnovan na algoritmu slučajne šume primenjen na programima sa takmičenja

bi trebalo da pokaže da li postoji ušteda u vremenu izvršavanja, bez drastičnog gubitka tačnosti, ako se koristi samo klasifikacioni model. Međutim, na osnovu dobijenih rezultata može se zaključiti da postojeći klasifikacioni model loše klasifikuje programe i iz tog razloga neće biti pokretan nijedan verifikacioni alat niti vršena poređenja.

Dobijeni rezultati, koliko god bili obeshrabrujući, imaju smisla. Pretpostavka je da su programi korišćeni za obučavanje modela zapravo preuzeti iz industrijskih aplikacija. Ovi programi su raznovrsni, ali su dati samo u obliku atributa te stoga njihov izgled ostaje nepoznat.

Programi korišćeni u evaluaciji su specijalno napravljeni za takmičenje u verifikaciji i kao takvi sadrže drugačije osobine i konstrukcije koje se ne mogu primeniti na programe iz klase koja je korišćena za obučavanje. Programi korišćeni za evaluaciju su prilično homogeni. Razlike između programa koji imaju i onih koji nemaju grešku su minimalne, kao na primer da jedan program ima znak $<$ a drugi $\leq$ (tj. $<=$ u kodu). Iz ovakvih razlika se ne može izvesti neki poseban zaključak osim da u slučaju znaka $<$ postoji greška, a u slučaju znaka $\leq$ greška ne postoji. Takve informacije se ne mogu smisleno generalizovati. Još jedan razlog za slabije ponašanje modela je priroda korišćenih atributa. Oni se zasnivaju na linijama koda, komentara, broju operatora i broju operanada. Iz tog ugla razlika između dva programa, onog sa greškom i onog bez greške, je u tome što jedan od programa sadrži jedan karakter $=$ više. Navedene razlike su previše suptilne da bi na osnovu datih podataka bile uhvaćene metodama mašinskog učenja pa je u takvim situacijama potrebno koristiti sofisticirane alate verifikacije softvera.

Glava 5

Zaključak

U ovom radu je opisan pokušaj poboljšanja efikasnosti verifikacije softvera korišćenjem metoda nadgledanog učenja. Predstavljen je postupak obučavanja, evaluacije i unapređenja klasifikacionih modela, čiji je zadatak da predvide da li program sadrži grešku u kodu ili ne, zasnovanih na algoritmima k najbližih suseda, logističke regresije i slučajne šume. Implementiran je program koji na osnovu izvornih kodova programa napisanih u programskom jeziku C izračunava atribute koji se koriste kao ulazi klasifikacionih modela. Program je upotrebljen da bi se iz korpusa programa korišćenih na takmičenju u verifikaciji izračunali atributi, nakon čega su izračunati atributi iskorišćeni kao ulaz u klasifikacioni model. Nažalost, rezultati su bili nezadovoljavajući pošto je model loše klasifikovao programe.

Dobijeni rezultati ne znače da se mašinskim učenjem ne može poboljšati efikasnost verifikacije. Naprotiv, postoji mnogo prostora za unapređenje rada. Korpus korišćen za obučavanje modela sadrži samo atribute programa. Pitanje je šta bi se dogodilo ako bi u fazi obučavanja bili dostupni čitavi programi. Sami atributi ne govore ništa o prirodi korišćenih programa, niti iz kojih oblasti su uzeti. Može se pretpostaviti da nisu svi programi podjednako pogodni za obučavanje modela. Pored korišćenih programa, moguće je i da korišćeni atributi nisu adekvatni za rešavanje ovog problema. Zbog toga bi bilo poželjno pokušati sa nekim drugim metrikama. Eksperimentisanje sa različitim vrednostima parametara prilikom obučavanja modela je jedna od opcija. U potrazi za boljim rezultatima trebalo bi isprobati i neke druge algoritme klasifikacije.

Bibliografija

[1] Foreword for „Programming Python" (1st ed.). https://legacy.python.org/doc/essays/foreword/.

[2] imbalanced-learn. https://imbalanced-learn.readthedocs.io/en/stable/index.html.

[3] jm1 dataset. http://promise.site.uottawa.ca/SERepository/datasets/jm1.arff.

[4] Matplotlib.org. https://matplotlib.org/.

[5] Nasa ground system. https://www.nasa.gov/exploration/systems/ground/index.html.

[6] pandas: powerful Python data analysis toolkit. https://pandas.pydata.org/pandas-docs/stable/.

[7] Preprocessing data. http://scikit-learn.org/stable/modules/preprocessing.html.

[8] pycparser. https://github.com/eliben/pycparser.

[9] Python Documentation by Version. https://www.python.org/doc/versions/.

[10] scikit-learn: Machine Learning in Python. http://scikit-learn.org/stable/index.html.

[11] Scipy.org. https://www.scipy.org/.

[12] SciPy.org: Quickstart tutorial. https://docs.scipy.org/doc/numpy/user/quickstart.html.

[13] sv-benchmarks. `https://github.com/sosy-lab/sv-benchmarks`.

[14] What is python? executive summary. `https://www.python.org/doc/essays/blurb/`.

[15] What's New In Python 3.0. `https://docs.python.org/3.0/whatsnew/3.0.html`.

[16] Y2k bug. `https://www.nationalgeographic.org/encyclopedia/Y2K-bug/`.

[17] Charu C. Aggarwal. *Data Mining: The Textbook*. Springer International Publishing, Switzerland, 2015.

[18] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press Cambridge, Massachusetts London, England, 2008.

[19] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.*, 51(3), 2018.

[20] Kaushal Bhatt, Vinit Tarey, and Pushpraj Patel. Analysis Of Source Lines Of Code(SLOC) Metric. *International Journal of Emerging Technology and Advanced Engineering*, 2(5), 2012.

[21] Jason Brownlee. *Machine Learning Mastery With Python: Understand Your Data, Create Accurate Models and Work Projects End-To-End*.

[22] Jason Brownlee. *Master Machine Learning Algorithms: Discover How They Work and Implement Them From Scratch*. 2016.

[23] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, Jul 2001.

[24] Lee Copeland. *A Practitioner's Guide to Software Test Design*. Artech House, 2004.

[25] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A Survey of Automated Techniques for Formal Software Verification. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7), 2008.

[26] Mladen Nikolić i Anđelka Zečević. Mašinsko učenje. 2018.

[27] Milena Vujošević Janičić. *Automatsko generisanje i proveravanje uslova ispravnosti programa*. PhD thesis, Matematički fakultet, Univerzitet u Beogradu, 2013.

[28] M.H. Halstead. *Elements of Software Science*. Elsevier, 1977.

[29] Glenford J. Myers. *The Art of Software Testing*. John Wiley & Sons, Inc., Hoboken, New Jersey, 2 edition, 2004.

[30] Indranil Nandy. HALSTEAD's Operators and Operands.

[31] Ron Patton. *Software Testing*. Sams Publishing, 800 E. 96th St., Indianapolis, Indiana, 46240 USA, 2001.

[32] J. Sayyad Shirabad and T.J. Menzies. The PROMISE Repository of Software Engineering Databases. School of Information Technology and Engineering, University of Ottawa, Canada, 2005.

[33] Prof. Dr. Holger Schlingloff. Software testing I. Humboldt-Universität zu Berlin and Fraunhofer Institute of Computer Architectureand Software Technology, November 2006.

[34] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014.

[35] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, Massachusetts, London, England, 2 edition, 2015.

[36] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Addison-Wesley Longman Publishing Co., Inc., 2005.

[37] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Publishers is an imprint of Elsevier. 500 Sansome Street, Suite 400, San Francisco, CA 94111, 2 edition, 2005.